HEWLETT ®
PACKARD

64-Bit Runtime Architecture and Software Conventions for IA-64

Version 2.5
July 16, 1999

IA-64 Runtime Architecture Task Force

Legal Notices

The information contained in this document is subject to change without notice.

Table of Contents

Chapter 1 Introduction 1

1.1 Objectives of the runtime architecture *1*

1.2 About the conventions *2*

1.3 Glossary *2*

1.4 Revision history *4*

Chapter 2 Processor Architecture 9

2.1 Application state and programming model *10*

2.2 Floating-point programming model *10*

2.3 System state and programming model *11*

2.4 Addressing and protection *11*

2.5 Interruptions *11*

Chapter 3 Memory Model 13

3.1 Program segments *14*

3.2 Protection areas *15*

3.3 Data allocation *16*

Chapter 4 Data Representation 19

4.1 Fundamental types *19*

4.2 Aggregate types *20*

4.3 Bit fields *23*

4.4 Fortran data types *25*

Chapter 5 Register Usage 27

5.1 Partitioning *27*

5.2 General registers *27*

5.3 Floating-point registers *29*

5.4 Predicate registers *29*

5.5 Branch registers *30*

5.6 Application registers *30*

Chapter 6 Register Stack 33

6.1 Input and local registers *33*

6.2 Output registers *34*

6.3 Rotating registers *34*

6.4 Frame markers *35*

6.5 Backing store for register stack *35*

Chapter 7 Memory Stack 37

7.1 Procedure frames *38*

Chapter 8 Procedure Linkage 41

8.1 External naming conventions *41*

8.2 The gp register *41*

8.3 Types of calls *41*

8.4 Calling sequence *42*

8.5 Parameter passing *46*

8.6 Return values *54*

8.7 Requirements for unwinding the stack *55*

Chapter 9 Coding Conventions 57

9.1 Sample code sequences *57*

9.2 Speculation *61*

9.3 Multi-threaded code *61*

9.4 Setjmp and longjmp *62*
9.5 Up-level referencing *62*
9.6 C++ conventions *63*

Chapter 10 Context Management 65
10.1 Process/thread context *65*
10.2 User-level thread switch, coroutines *66*
10.3 Setjmp/longjmp *67*

Chapter 11 Stack Unwinding and Exception Handling 69
11.1 Unwinding the stack *70*
11.2 Exception handling framework *71*
11.3 Coding conventions for reliable unwinding *73*
11.4 Data structures *77*

Chapter 12 Dynamic Linking 89
12.1 Position-independent code *89*
12.2 Import stubs *90*
12.3 The dynamic loader *91*

Chapter 13 System Interfaces 93
13.1 Program startup *93*
13.2 System calls *94*
13.3 Traps and signals *94*

Appendix A Standard Header Files 95
A.1 Implementation Limits *95*
A.2 Floating-Point Definitions *95*
A.3 Variable Argument List Macros *96*
A.4 Setjmp/Longjmp *97*

Appendix B Unwind Descriptor Record Formats 99

B.1 Overview *99*

B.2 Region Header Records *100*

B.3 Descriptor Records for Prologue Regions *101*

B.4 Descriptor Records for Body Regions *106*

B.5 Descriptor Records for Body or Prologue Regions *107*

Chapter 1

Introduction

This document describes common software conventions for the Enhanced Mode extensions to the Intel IA-64 Architecture. It does not define operating-system interfaces or any conventions specific to any single operating system.

The runtime architecture defines most of the conventions necessary to compile, link, and execute a program on an operating system that supports these conventions. Its purpose is to ensure that object modules produced by different compilers can be linked together into a single application, and to specify the interfaces between compilers and linker, and between linker and operating system.

The runtime architecture does not specify the Application Programming Interface (API), the set of services provided by the operating system to the program, nor does it specify certain conventions that are specific to each operating system. Thus, conformance to the runtime architecture alone is not sufficient to produce a program that will execute on all IA-64 platforms. It does, however, allow many of the development tools to be shared among various operating systems.

When combined with the instruction set architecture, an API, and system-specific conventions, this runtime architecture leads to an Application Binary Interface (ABI). In other words, an ABI is the composition of an API, system-specific conventions, a hardware description, and a runtime architecture.

1.1 Objectives of the runtime architecture

This document defines the software interfaces needed to ensure that software for IA-64 will operate correctly together. The intent is to define as small a set of interface specifications as possible, while still meeting the following goals:

- Support 64-bit addressing and datatypes
- High performance
- Ease of porting
- Ease of interfacing with IA-32 and PA-RISC
- Ease of implementation and use
- Complete enough to insure software compatibility

We would like to provide complete enough interfaces between the different software products that they can be provided by different ISVs and still work together. These include compilers, linkers, applications, and dynamic link libraries. The goal is to have one standard, so software will be portable on IA-64 systems.

1.2 About the conventions

ANSI C serves as the reference programming language. By defining the implementation of C data types, the software conventions can give precise system interface information without resorting to assembly language. Giving C language bindings for system services does *not* preclude bindings for other programming languages. Moreover, the examples given here are not intended to specify the C language available on the system.

1.3 Glossary

The following terms are used in this document:

Absolute address. In this document, the term absolute address refers to a virtual address, not a physical address. It is an address within the process' address space that is computed as an absolute number, without the use of a base register.

Binding. The process of resolving a symbolic reference in one module by finding the definition of the symbol in another module, and substituting the address of the definition in place of the symbolic reference. The linker binds relocatable object modules together, and the DLL loader binds executable load modules. When searching for a definition, the linker and DLL loader search each module in a certain order, so that a definition of a symbol in one module has precedence over a definition of the same symbol in a later module. This order is called the **binding order**.

Dynamic-link library (DLL). A library that is prepared by the linker for quick loading and binding when a program is invoked, or while the program is running. A DLL is designed so that its code is shared by all processes that are bound to it. (Also called **shared library**.)

Execution time. The time during which a program is actually executing, not including the time during which it and its DLLs are being loaded.

Function pointer. A reference or pointer to a function. A function pointer takes the form of a pointer to a special descriptor (a **function descriptor**) that uniquely identifies the function. The function descriptor contains the address of the function's actual entry point as well as its global data pointer (gp).

Global data pointer (gp). The address of a reference location in a load module's data segment, usually kept in a specified general register during

execution. Each load module has a single such reference point, typically near the middle of the load module's linkage table. Applications use this pointer as a base register to access linkage table entries, and data that is local to the load module.

Link time. The time when a program or DLL is processed by the linker. Any activity taking place at link time is static.

Linkage table. A table of addresses that contains pointers to code or data that is external to the load module, or that cannot be addressed directly. Each load module contains a linkage table in its data segment, which allows external references to be bound dynamically without modifying the application's code.

Load module. An executable unit produced by the linker, either a main program or a DLL. A program consists of at least a main program, and may also require one or more DLLs to be loaded to satisfy its dependencies.

Own data. Data belonging to a load module that is referenced directly from that load module and that is not subject to the binding order. If a module references a data item symbolically, and another module earlier in the binding order defines an item with the same symbolic name, the reference is bound to the data item in the earlier module. If this is the case, the data is not "own." Typically, own data is local in scope.

PC-relative addressing. Code that uses its own address (commonly called the program counter, or "PC"; this is called the instruction pointer, or IP, in the IA-64 architecture) as a base register for addressing other code and data.

Position-independent code (PIC). This term has a dual meaning. First, position-independent code is designed so that it contains no dependency on its own load address; usually, this is accomplished by using pc-relative addressing so that the code does not contain any absolute addresses. Second, it also implies that the code is also designed for dynamic binding to global data; this is usually done by using indirect addressing through a linkage table.

Preserved register. A register that is guaranteed to be preserved across a procedure call.

Program invocation time. The time when a program or DLL is loaded into memory in preparation for execution. Activities taking place at program invocation time are generally performed by the system loader or dynamic loader.

Protection area. A portion of a segment that shares common access protections.

Region. The IA-64 architecture divides the address space into four or eight regions. In general, the runtime architecture is independent of which segments are assigned to which region.

Scratch register. A register that is not preserved across a procedure call.

Segment. An area of memory that has specific attributes, and behaves as a fixed unit at runtime. All items within a segment have a fixed address relationship to one another at execution time, and have a common set of attributes. Items in different segments do not necessarily bear this relationship, and an application may not depend on one. For example, the program text segment is defined to contain the main program code, unwind information, and read-only data. The use of this term is not related to the concept of a segment in the IA-32 architecture, nor is it directly related to the concept of a segment in an object file.

Static. (1) Any data or code object that is allocated at a fixed location in memory and whose lifetime is that of the entire process, regardless of its scope; (2) A binding that takes place at link time rather than program invocation or execution time.

1.4 Revision history

Changes in Version 2.1

The conventions have been updated in general to reflect the differences between Versions 1.0 and 2.1 of the EAS.

Specifically, this revision includes the following major changes relative to the previously-published version (Version 1.0):

- Chapter 2 now includes conventions for privilege level, probe instructions, and break instructions.
- Chapter 3 has been revised with the concept of protection areas.
- In Chapter 4, the data representation has been changed to an LP64 model, rather than ILP64, and the long double type has been removed.
- In Chapter 5, the floating-point registers, branch registers, and predicate registers have been repartitioned. The branch registers and predicates were repartitioned to reflect the change in the branch architecture and the increase in number of predicates. ar.k3 has been designated for use as a thread pointer.
- Chapter 8 has been revised substantially. The return link is now b0, and small integer scalars are no longer sign- or zero-extended when passed as parameters.
- Chapter 9 now describes example coding conventions. Some of the language-specific material was moved to other chapters.
- Chapter 11 has been substantially rewritten, and now describes the stack unwind table format (along with the new Appendix B).
- Two appendices have been added. The first contains standard definitions from limits.h, float.h, stdarg.h, and setjmp.h header files; the second contains the detailed descriptions of the stack unwind table formats.

Changes in Version 2.3

The conventions have been updated in general to reflect the differences between Versions 2.1 and 2.3 of the EAS.

Specifically, this revision includes the following major changes relative to the previously-published version (Version 2.1):

- In Chapter 1, the terms “function pointer” and “function descriptor” have replaced the older terms “procedure pointer/plabel” and “plabel descriptor.”
- In Chapter 2, a list of architected software interrupts, for use with the break instruction, has been added, and the conventions for use of the probe instructions have been modified so that a failure result from a probe instruction is not definitive.
- References to the non-PIC programming model were removed from Chapter 3, since non-PIC is ABI- or implementation-specific.
- In Chapter 4, the long long data type was removed, and 128-bit integer and floating-point types have been added (although implementation of these types is not required).
- In Chapter 5, the conventions for gp were modified slightly to work better with (ABI-specific) non-PIC code, the thread pointer was moved from ar.k3 to r13, and the conventions for the floating-point status register were modified.
- Chapter 7 now has an expanded discussion of dynamic allocation in the memory stack frame.
- In Chapter 8, conventions for passing and returning 128-bit integers and 128-bit floating-point arguments were added, and rationale was added for the treatment of NaT bits and NaT-vals on incoming parameters.
- In Chapter 9, the suggested conventions for up-level referencing were expanded, and a C++ convention for copy constructors was added.
- In Chapter 10, the conventions for user-level thread switches and setjmp/longjmp were updated to handle the ALAT correctly; the RSE NaT collection register was removed from the list of items in the saved state for setjmp/longjmp.
- A number of new unwind descriptor records were added in Chapter 11 to handle additional preserved registers. The layout of the spill area was modified to make better use of the User NaT collection register.
- Chapter 12 was updated to remove obsolete and incorrect references to non-PIC code, which is not allowed by these conventions (even in main programs).
- An initial value for the floating-point status register is specified in Chapter 13.
- In Appendix A, the jump buffer alignment was increased and its size has been made ABI specific.
- In Appendix B, the record types for formats P3 and P7 were renumbered, and formats P8 and P9 were added.

Changes in Version 2.4

- In Chapter 3, the table of protection areas was modified to show the linkage tables as writable.
- Chapter 4 was revised to allow the use of either the “LP64” or “P64” data model. Thus, the definition of the “long” data type is dependent on the ABI. Throughout this document, “__int64” is used to refer to the 64-bit integer data type.
- In Chapter 4, the conventions for enumerated types were modified to match existing practice in most IA-32 compilers. Enumerated base types are always signed; bit fields of enumerated types are unsigned only if necessary to represent the enumeration constants within the designated width.
- In Chapter 5, the conventions for the gp register were modified slightly to allow a compiler to optimize calls known to branch to functions in the same load module.
- In Chapter 8, the conventions for passing quad-precision floating-point numbers were changed. When passed in registers, quad-precision floating-point values are now passed in general registers instead of floating-point registers.
- In Chapter 8, the conventions for returning integers smaller than 32 bits were modified to specify that the values should be zero- or sign-extended (rather than padded with garbage).
- In Chapter 9, a note about speculative loads to unaligned addresses was added, to point out that without recovery code, the application cannot expect to emulate an unaligned load.
- In Chapter 9, conventions for the use of the volatile keyword in ANSI C were added.
- In Chapter 10, a distinction was added between synchronous and asynchronous context switches.
- In Chapter 10, the procedures for switching the register stack were updated.

Changes in Version 2.5

- In Chapter 8, the alignment rules for passing aggregates by value were changed, so that the alignment in the parameter list is a function of the external alignment of the aggregate rather than a function of the size of the aggregate.
- In Chapter 8, the rules for passing floating-point parameters in general registers were reworded to make it clearer that the behavior is dependent on the compiler’s knowledge of the formal parameter list, rather than specifically on a C or C++ function prototype.
- In Chapter 8, two additional examples of passing aggregates by value have been provided.
- In Chapter 10, the table of resources required to be saved on a synchronous context switch was missing two registers; it has been corrected.

- In Chapter 10, the procedures for switching the register stack in a context switch were updated to match the IA-64 architecture document.
- In Chapter 11, a set of bits from the unwind information block header word has been reserved for vendor-specific use.
- In Chapter 11, several new unwind descriptor records have been added to support arbitrary spill and restore sequences, as well as predicated spills.
- In Chapter 13, a rule was added that the floating-point status register must be initialized to a defined value when a signal is delivered to a process.
- In Appendix A, the prefixes for double-extended precision and quad precision constants were changed.
- In Appendix A, the variable argument list macros were changed for big-endian environments to reflect the change in rules for passing aggregates by value.
- In Appendix B, a new table summarizing the descriptor encodings has been added, and the formats for the new unwind descriptors have been added.

Chapter 2

Processor Architecture

It is assumed that applications conforming to this specification will run in a software environment provided by some operating system, and that additional conventions will be specified as part of the Application Binary Interface (ABI) for that operating system. It is further assumed that the operating system will restrict the application's access to the physical resources of the machine, by limiting the privilege level of the application and by using virtual memory to define the address space available to the application.

The *Enhanced Mode External Architecture Specification (EAS)*, *Version 2.5*, defines the IA-64 instruction set architecture. Programs intended to execute directly on an IA-64 processor use the instruction set, instruction encodings, and instruction semantics defined in the EAS. Three points deserve explicit mention:

- A program may assume all documented instructions exist.
- A program may assume all documented instructions work.
- A program may use only the instructions defined by the architecture.

In other words, from a program's perspective, the execution environment provides a complete and working implementation of IA-64.

This does not imply that the underlying implementation provides all instructions in hardware, only that the instructions perform the specified operations and produce the specified results. The software conventions neither place performance constraints on systems nor specify what instructions must be implemented in hardware. A software emulation of the architecture could conform to these conventions.

Some processors might support IA-64 as a subset, providing additional instructions or capabilities. Programs that use those capabilities explicitly do not conform to these conventions. Executing those programs on machines without the additional capabilities results in undefined behavior.

These conventions are intended for application use, and so use only features found in user mode. Applications should assume that they will execute in user mode (privilege level 1, 2, or 3), and that any attempt to use processor resources restricted to privilege level 0 will cause a trap that may terminate the process.

2.1 Application state and programming model

An application may use all features of IA-64 that are described in the Application State and Programming Model section of the EAS.

Application use of the **break** instruction is subject to the following conventions:

- Immediate operands whose three highest-order bits are 000 are reserved for architected software interrupts. These software interrupts are listed in Table 2–1. Application programs (typically language runtime support libraries) may check for these conditions and raise these interrupts, but are not required to do so. Immediate operands in this range, and not listed in the table, are reserved for future use.
- Immediate operands whose three highest-order bits are 001 are available for application use as software interrupts. The behavior of these interrupts, however, is ABI specific.
- Immediate operands whose two highest-order bits are 01 are reserved for debugger breakpoints. Use of debugger breakpoints is ABI specific.
- Immediate operands whose highest-order bit is 1 are reserved for definition by each ABI. It is expected that some operating systems may use values in this range for system-level debugging features and system calls.

Table 2–1. Software Interrupts

Operand	*Software Interrupt*
0	Unknown program error (typically an indirect branch through an uninitialized pointer, which often leads to a bundle containing all zeroes)
1	Integer divide by zero
2	Integer overflow
3	Range check/bounds check error
4	Nil pointer dereference
5	Misaligned data
6	Decimal overflow
7	Decimal divide by zero
8	Packed decimal error
9	Invalid ASCII digit (unpacked decimal arithmetic)
10	Invalid decimal digit (packed decimal arithmetic)
11	Paragraph stack overflow (COBOL)

2.2 Floating-point programming model

An application may use all features of the processor architecture that are described in the Floating-Point Programming Model section of the EAS.

2.3 System state and programming model

The features of the processor architecture that are described in the System State and Programming Model section of the EAS are intended for the exclusive use of the operating system software, with the following exceptions:

- The Interval Time Counter application register may be read by applications, except when running in a secure operating environment that explicitly restricts this access.
- The explicit serialization instructions may be used by an application.
- An application may read and modify the user mask portion of the PSR, although some changes may result in unexpected and incorrect interactions with the operating system software. Changes to the user mask should be done only as allowed by the ABI.
- An application may use the RSE-related instructions, and may read and modify the resources associated with the register stack engine that are not restricted to privilege level 0.

Note that the debug and performance monitor control registers are restricted for use by the operating system software, which may provide access to the capabilities provided by these hardware features through its APIs. Although the performance monitor counter registers are readable by user-mode code, effective use of the registers is dependent on ABI-specific services.

2.4 Addressing and protection

The features of the processor architecture that are described in the Addressing and Protection section of the EAS are intended for the exclusive use of the operating system software, with the following exceptions:

- An application may use the addp4 and shladdp4 instructions to convert a 32-bit virtual address to a 64-bit virtual address.
- The operating system software may provide access to certain page attributes, including caching and ordering attributes, through its API. The use of such features is ABI specific.
- Applications may use the probe instructions, but a failure result does not necessarily indicate a lack of permission. In particular, a probe for write access to a copy-on-write page is not guaranteed to return a success result. The operating system software is permitted to nullify a faulting probe instruction, so application software must pre-initialize the target register in order to distinguish a success result from a nullified probe instruction.

2.5 Interruptions

The features of the processor architecture that are described in the Interruptions section of the EAS are intended for the exclusive use of the operating system software.

Chapter 3

Memory Model

These conventions define a virtual memory system with a 64-bit virtual address space per process. Each operating system may divide this address space into different portions, and assign specific uses to each portion.

This chapter describes the types of memory segments and protection areas that an application process uses, and documents the assumptions that an application may make about those segments. From a different perspective, it documents the minimum requirements that must be satisfied by an operating system with respect to its allocation of these program segments in the virtual address space.

The term *segment* is used here to identify an area of memory that has a specific use within an application and has no fixed address relationship to any other segment. Thus, relative distances between any two items belonging to the same segment are constant once the program has been linked, but the distance between two items in different segments is not fixed. It does not imply the use of hardware segmentation, or any specific allocation of segments to hardware regions. In particular, this definition of segment has no relation to the traditional IA-32 segment, nor does it necessarily correspond exactly to the definition of a segment in an object file.

Segments may cross region boundaries. Region ids should be transparent to the application. Note that more than one region register may point to the same region.

Segments are composed of one or more protection areas. The term *protection area* is used to indicate an area of memory that has common protection attributes.

3.1 Program segments

Table 3–1 lists the types of program segments that are defined by the runtime architecture, and defines the minimum set of attributes that an operating system must provide for these segments.

Table 3–1. Program Segments

Segment Type	*Sharable*	*Quantity*	*Address by*	*Contents*
Text	Yes	1 per load module	IP or linkage table	Text, unwind information, constants and literals
Short Data	No	1 per load module	gp	Static data, bss, linkage tables
Long Data	No	any	linkage table	Long data, bss
Heap	No	any	pointer	Heap data
Stack	No	1 per thread	sp	Memory stacks
Backing store	No	1 per thread	bsp	Backing store for register stacks
Thread data	No	1 per thread	tp	Thread-local storage
Shared data	Yes	any	pointer	Shared memory

The sharable attribute indicates whether or not the memory contained within such a segment may be shared between two or more processes. For text segments, this implies that an OS will probably not grant write access, in order to make the text segment pure. For this reason, the runtime architecture does not place anything into the text segment that may need to be written at either program invocation time or execution time.

The runtime architecture does not specify how an OS will make a particular segment sharable. It may place sharable segments in separate regions, or it may place the entire program in a process-private address space and use address aliasing to share memory. The runtime architecture is designed to be neutral with respect to this OS design parameter. Segments may cross hardware region boundaries, but only if transparent to the application. Code is not aware of region IDs.

A program consists of several load modules: the main program, and one for each DLL that it uses. Each load module consists of at least a text segment and a short data segment. The addresses of these segments are not fixed at link time, so all accesses to these segments must be either pc-relative (for text), gp-relative (for short data and the linkage table), or indirect via the linkage table. The `gp` register and its conventions are described in Chapter 8.

DLL data may be allocated at execution time. This implies that DLL data segment sizes need not be fixed at linkage time.

Each OS is expected to provide some form of heap management, although the runtime architecture does not have any explicit dependencies on such. The API for obtaining heap memory, however, is OS dependent, and the runtime architecture places no restrictions on the locations or contiguity of separately-allocated items from the heap.

Each thread is provided with two stacks: one for the classical memory stack, and one for the register stack backing store. Each thread also has a separate data segment for thread-local storage. These segments must all be allocated from the process' virtual address space, so that one thread may use a pointer that refers to another thread's local storage. The `sp` register and its conventions are described in Chapter 7, and the `bsp` register is described in Chapter 6. The `tp` register is reserved to provide a handle for accessing thread-local storage, but this usage is ABI dependent.

Like the heap, shared data segments are obtained through an OS-specific API. The runtime architecture places no restrictions on the locations of these segments.

3.2 Protection areas

Table 3–2 lists the minimum access protection for the protection areas defined in the runtime architecture:

Table 3–2. Protection Areas

Segment	*Protection Area*	*Min. Access*
Text	Text	X
	Constants	R
	Unwind Tables	R
Short data	Static Data	R, W
	Short Bss	R, W
	Linkage Tables	R, W
Long data	Long Data	R, W
	Bss	R, W
Heap	Heap	R, W
Stack	Stack	R, W
Backing store	Backing store	R, W
Thread data	Thread data	R, W
Shared data	Shared data	R, W

In order to make the most effective use of the addressing modes available in IA-64, each load module's data is partitioned into one short and some number of long data segments. The short data segment, addressed by the `gp` register in each load module, contains the following areas:

- A linkage table, containing pointers to imported data symbols and functions, and to data in the text segments and long data segments.
- A short data area, containing small initialized "own" data items.
- A short bss area, containing small uninitialized "own" data items.

The long data segments contain either or both of the following areas:

- A long data area, containing large initialized data items, and initialized non-"own" data items of any size.

- A long bss area, containing large uninitialized data items, and uninitialized non-“own” data items of any size.

“Own” data items are those that are either local to a load module, or are such that all references to these items from the same load module will always refer to these items. That is, they are not subject to being overridden by an exported symbol of the same name in another load module. All data items in the main program satisfy this definition, since the main program is always the first load module in the binding sequence. Since non-“own” variables cannot be referenced directly, there is no benefit to placing them in the short data or bss area.

Small “own” data items are placed in the short bss or short data, and are guaranteed to be within 2 megabytes, in either direction, of the `gp` address, so compilers may use a short direct addressing sequence (using the add with 22-bit immediate instruction) to access any data item allocated in these areas. The compiler should place all “own” data items that are 8 bytes or less in size, regardless of structure, in the short data or short bss areas.

All other data items, including items that are larger than 8 bytes in size, or that require indirect addressing because of load-time binding, must be placed in the long data or long bss area. The compiler must address these items indirectly, using a linkage table entry. Linkage table entries are typically allocated by the linker in response to a relocation request generated by the compiler; an entry in the linkage table is either an 8-byte pointer to a data item, or a 16-byte function descriptor. A function descriptor placed in the linkage table is a local copy of an “official” function descriptor that is generally allocated by the linker or dynamic loader.

This design allows for a maximum size of 4 megabytes for the short data segment, since everything must be addressable via the `gp` register using the 22-bit add immediate instruction. Given that linkage table entries are 8 byte pointers for data references, and 16 bytes long for procedure references, this allows for up to 256,000 individually-named variables and functions. If a load module requires more than this, the compilers will need to support a “huge” memory model, which is not described here.

Protection areas are required to be aligned only as strictly as their contents.

3.3 Data allocation

3.3.1 Global variables

Common blocks, dynamically allocated regions (such as `malloc`, etc.), and external data items greater than 8 bytes must all be aligned on a 16-byte boundary. Smaller data items must be aligned on the next larger power-of-two boundary. Table 3–3 shows the alignment requirements for different size objects.

Table 3–3. Alignment Requirements for Global Objects

Size in bytes	*Alignment required*
1	none
2	0 mod 2 (even addresses)
3–4	0 mod 4
5–8	0 mod 8
9 and up	0 mod 16

Access to global variables that are not known (at compile time) to be defined in the same load module must be indirect. Each load module has a linkage table in its data segment, pointed to by the `gp` register; code must load a pointer to the global variable from the linkage table, then access the global variable through the pointer. Access to globals known to be defined in the same load module or to static locals may be made with a gp-relative offset.

3.3.2 Local static data

Access to local static data can be made with a gp-relative offset.

3.3.3 Constants and literals

Constants and literals may be placed in the text segment or in the data segment. If placed in the text segment, the access must be pc-relative or indirect using a linkage table entry.

Literals placed in the data segment may be placed in the short initialized data area if they are 8 bytes or less in size. Larger literals must be placed in the long initialized data area or in the text segment. Literals in the long initialized data area require an indirect access using a linkage table entry.

3.3.4 Local memory stack variables

Access is sp-relative.

Stack frames must always be aligned on a 16-byte boundary. That is, the stack pointer register must always be aligned on a 16-byte boundary.

Chapter 4

Data Representation

Applications running in a 64-bit environment use either the “P64” or “LP64” data model: integers are 32 bits, while pointers are 64 bits. Long integers may be either 32 or 64 bits, depending on the ABI.

Within this specification, the term *halfword* refers to a 16-bit object, the term *word* refers to a 32-bit object, the term *doubleword* refers to a 64-bit object, and the term *quadword* refers to a 128-bit object.

The following sections define the size, alignment requirements, and hardware representation of the standard C and Fortran datatypes.

Note *IA-64 will support misaligned access, at a substantial penalty. Some implementations may emulate mis-alignment support by trapping. These alignment rules are chosen to maximize performance.*

4.1 Fundamental types

Table 4–1 lists the scalar datatypes supported by the architecture. Sizes and alignments are shown in bytes.

Table 4–1. Scalar Datatypes Supported by IA-64

Type	*C*	*Size*	*Alignment*	*Hardware Representation*
Integral	char signed char	1	1	signed byte
	unsigned char	1	1	unsigned byte
	short signed short	2	2	signed halfword
	unsigned short	2	2	unsigned halfword
	int signed int enum	4	4	signed word
	unsigned int	4	4	unsigned word
	__int64 signed __int64	8	8	signed doubleword
	unsigned __int64	8	8	unsigned doubleword

Table 4–1. Scalar Datatypes Supported by IA-64

Type	*C*	*Size*	*Alignment*	*Hardware Representation*
	__int128 [1] signed __int128 [1]	16	16	signed 128-bit integer
	unsigned __int128 [1]	16	16	unsigned 128-bit integer
Pointer	*any-type* * *any-type* (*) ()	8	8	unsigned doubleword
Floating-point	float	4	4	IEEE single precision
	double	8	8	IEEE double precision
	__float80 [2]	16	16	IEEE double-extended precision
	__float128 [3]	16	16	quad precision

A null pointer (for all types) has the value zero.

Notes

[1] __int128 is not directly supported by the hardware, and these conventions do not require an operating system environment to support this type through emulation. Size and alignment conventions are specified here, however, for those implementations that do choose to support this type. Note also that the (non-standard) long long data type is not specified by these conventions, and may be implemented as a 64-bit integer, a 128-bit integer, or not at all.

[2] __float80 is the IA-64 extended 80-bit quantity, but the software standard is to treat it as a 16-byte quantity. It is referenced using ldfe and stfe instructions. This type has the same precision and range as the 80 bit extended datatype of the IA-32 architecture, but with different size and alignment.

[3] __float128 is not directly supported by the hardware, and these conventions do not require an operating system environment to support this type through emulation. Size, representation, and alignment conventions are specified here, however, for those implementations that do choose to support this type. A quad-precision floating-point number is a 128-bit quantity with a sign bit, a 15-bit biased exponent, and a 112-bit mantissa with an implicit integer bit.

4.2 Aggregate types

Aggregates (structures and arrays) and unions assume the alignment of their most strictly aligned component. The size of any object, including aggregates and unions, is always a multiple of the object's alignment. An array uses the same alignment as its elements. Structure and union objects can require padding to meet size and alignment constraints. The contents of any padding is undefined.

- An entire structure or union object is aligned on the same boundary as its most strictly aligned member.

- Each member is assigned to the lowest available offset with the appropriate alignment. This may require *internal padding*, depending on the previous member.
- A structure's size is increased, if necessary, to make it a multiple of the alignment. This may require *tail padding*, depending on the last member.

In the following figures, members' byte offsets appear in the upper right corners for little-endian, in the upper left for big-endian.

```
struct {
    char   c;
};
```

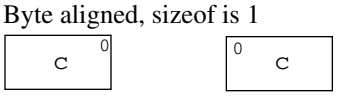


Figure 4–1. Structure Smaller Than a Word

```
struct {
    char   c;
    char   d;
    short  s;
    int n;
};
```

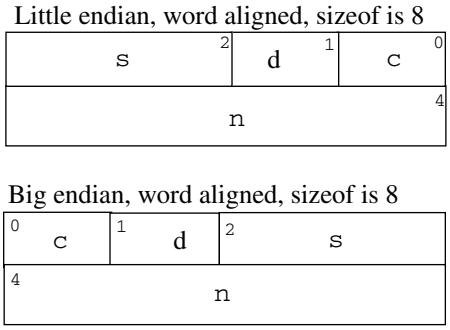


Figure 4–2. No Padding

```
struct {
    char   c;
    short  s;
};
```

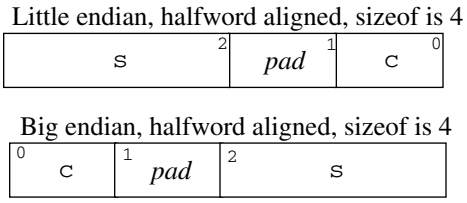
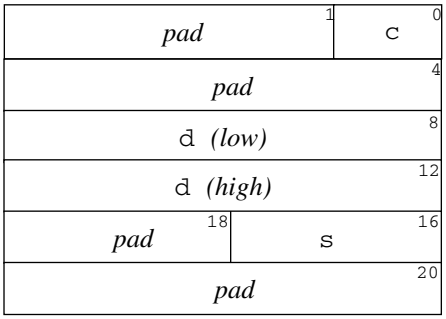


Figure 4–3. Internal Padding

```
struct {
    char   c;
    double d;
    short  s;
};
```

Little endian, doubleword aligned, sizeof is 24



Big endian, doubleword aligned, sizeof is 24

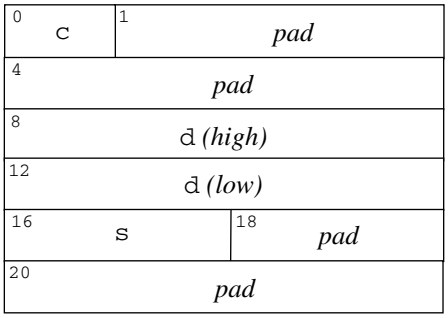
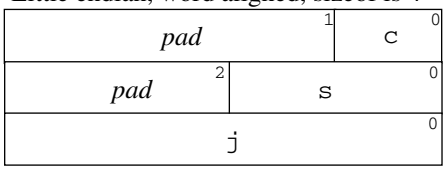


Figure 4–4. Internal and Tail Padding

```
union {
    char   c;
    short  s;
    int j;
};
```

Little endian, word aligned, sizeof is 4



Big endian, word aligned, sizeof is 4

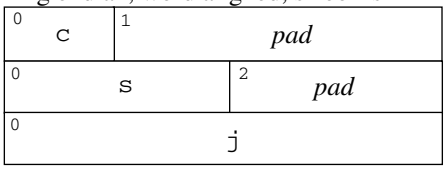


Figure 4–5. Union Allocation

4.3 Bit fields

C struct and union definitions may have *bit-fields* that define integral objects with a specified number of bits. Table 4–2 defines the allowable widths and corresponding range of values for bit fields of each base type.

Table 4–2. Bit Field Base Types

Base Type	*Width w*	*Range*
unsigned char	1 to 8	0 to 2^w–1
signed char	1 to 8	-2^{w-1} to 2^{w-1}–1
unsigned short	1 to 16	0 to 2^w–1
signed short	1 to 16	-2^{w-1} to 2^{w-1}–1
unsigned int	1 to 32	0 to 2^w–1
signed int	1 to 32	-2^{w-1} to 2^{w-1}–1
unsigned long	1 to 64	0 to 2^w–1
signed long	1 to 64	-2^{w-1} to 2^{w-1}–1

Bit-fields obey the same size and alignment rules as other structure and union members, with the following additions:

Byte Order

- Bit-fields are allocated from right to left (least to most significant) for little endian. They are allocated left to right (most to least significant) for big-endian.
- A bit-field must entirely reside in a storage unit appropriate for its declared type. For example, a bit field of type short must never cross a halfword boundary.
- Bit-fields may share a storage unit with other struct/union members, including members that are not bit-fields. Of course, each struct member occupies a different part of the storage unit.
- Unnamed bit-fields' types do not affect the alignment of a structure or union. Zero-length unnamed bit-fields force the aligment of subsequent members to the boundary corresponding to the size of the unnamed bit-field.

The following figures show struct and union member byte offsets in the upper corners; bit numbers appear in the lower corners.

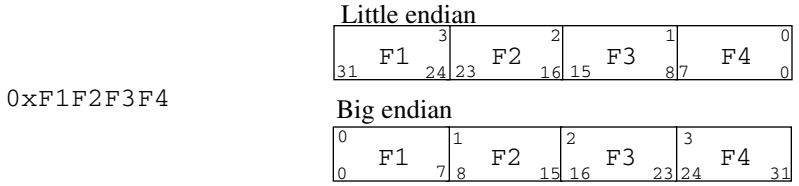
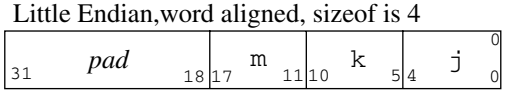


Figure 4–6. Bit Numbering

```
struct {
    int    j:5;
    int    k:6;
    int    m:7;
};
```

Little Endian,word aligned, sizeof is 4



Big Endian, word aligned, sizeof is 4

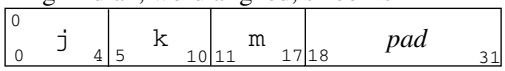
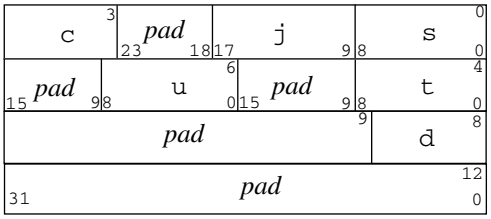


Figure 4–7. Bit Field Allocation

```
struct {
    short   s:9;
    __int64 j:9;
    char    c;
    short   t:9;
    short   u:9;
    char    d;
};
```

Little Endian, doubleword aligned, sizeof is 16



Big Endian, doubleword aligned, sizeof is 16

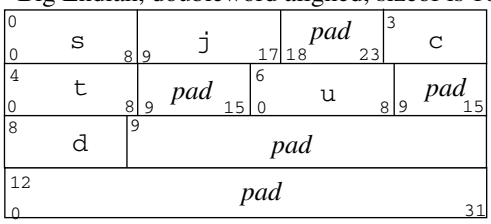
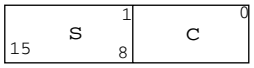


Figure 4–8. Boundary Alignment

```
struct {
    char   c;
    short  s:8;
};
```

Little-endian, halfword aligned, sizeof is 2



Big-endian, halfword aligned, sizeof is 2

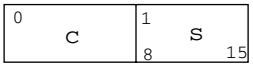


Figure 4–9. Storage Unit Sharing

```
union {
    char   c;
    short  s:8;
};
```

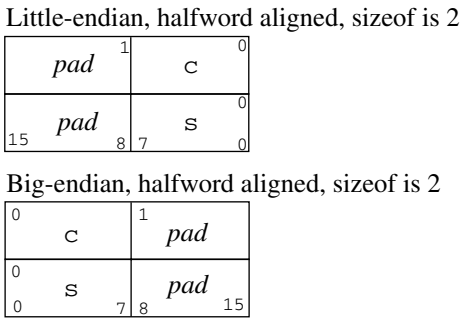


Figure 4–10. Union Allocation

```
struct {
    char   c;
    int    :0;
    char   d;
    short  :9;
    char   e;
    char   :0;
};
```

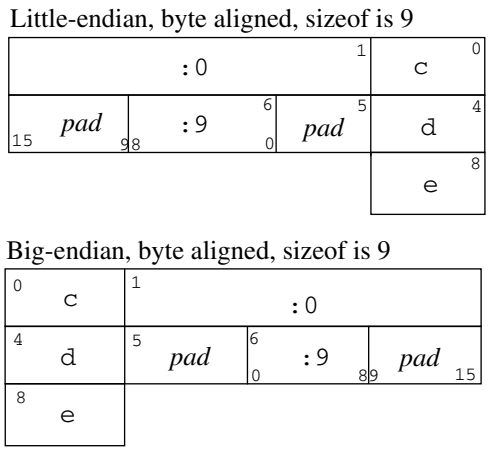


Figure 4–11. Unnamed Bit Fields

Note that unnamed bit fields do not affect the alignment of the structure.

As the examples show, int and __int64 bit-fields (including signed and unsigned) usually pack more densely than smaller base types. One can use char and short bit-fields to force allocation within those types, but int is generally more efficient.

4.4 Fortran data types

Table 4–3 shows the correspondence between ANSI Fortran's scalar types and the processor's data types. ANSI Fortran requires REAL and INTEGER to be the same size. Many Fortran compilers allow INTEGER*n, LOGICAL*n, and REAL*n to specify specific processor sizes. ("n" is in bytes). The COMPLEX datatype is treated exactly the same as a C structure composed of two float members.

Table 4–3. Fortran Data Types

Type	*Fortran*	*Size*	*Alignment (bytes)*	*Hardware Representation*
Character	CHARACTER*_n_	_n_	1	byte
Integral	LOGICAL	4	4	word
	INTEGER	4	4	signed word
Floating-point	REAL	4	4	IEEE single-precision
	DOUBLE PRECISION	8	8	IEEE double-precision
	COMPLEX	8	4	2 IEEE single-precision

Chapter 5

Register Usage

5.1 Partitioning

Registers are partitioned into the following classes:

- **Scratch** registers may be destroyed by a procedure call; the caller must save these registers before a call if needed (“caller-save”).
- **Preserved** registers must not be destroyed by a procedure call; the callee must save and restore these registers if used (“callee-save”).
- **Automatic** registers are saved and restored automatically by the call/return mechanism.
- **Constant** or **Read-only** registers contain a fixed value that cannot be changed by the program.
- **Special** registers are used in the call/return mechanism. The conventions for these registers are described individually below.

5.2 General registers

General registers are used for integer arithmetic and other general-purpose computations. Table 5–1 lists the general registers.

Table 5–1. General Registers

Register	*Class*	*Usage*
r0	constant	Always 0
r1	special	Global data pointer (gp)
r2–r3	scratch	Use with 22-bit immediate add
r4–r7	preserved	
r8–11	scratch	Procedure return value
r12	special	Memory stack pointer (sp)
r13	special	Reserved as a thread pointer (tp)
r14–r31	scratch	
in0–in95	automatic	Stacked input registers (see below)

Table 5–1. General Registers

Register	*Class*	*Usage*
loc0–loc95	automatic	Stacked local registers (see below)
out0–out95	scratch	Stacked output registers (see below)

- **r1** is the global data pointer (gp), which is designated to hold the address of the currently addressable global data segment. Its use is subject to the following conventions:

  a. On entry to a procedure, gp is guaranteed valid for that procedure.

  b. At any direct procedure call, gp must be valid (for the caller). This guarantees that an import stub (see Section 8.4.1) can access the linkage table.

  c. Any procedure call (indirect or direct) may destroy gp—unless the call is known to be local to the load module.

  d. At procedure return, gp must be valid (for the returning procedure). This allows the compiler to optimize calls known to be local (i.e., the exceptions to Rule 'c').

  The effect of these rules is that gp must be treated as a scratch register at a point of call (i.e., it must be saved by the caller), and it must be preserved from entry to exit.

- **r4–r7** are general-purpose preserved registers, and can be used for any value that needs to be preserved across a procedure call. A procedure using one of the preserved general registers must save and restore the caller's original contents, including the NaT bits associated with the registers, without generating a NaT consumption fault. This can be done by either copying the register to a stacked register or by using the st8.spill and ld8.fill instructions and then saving ar.unat.

- **r8** is used as the struct/union return pointer register. If the function being called returns a struct or union value larger than 32 bytes, then register GR 8 contains, on entry, the appropriately-aligned address of the caller-allocated area to contain the value being returned. (See Section 8.6.)

- **r8–r11** are used for non-floating-point return values up to 32 bytes. Functions do not have to preserve their values for the caller.

- **r12** is the *stack pointer*, which holds the limit of the current stack frame, the address of the stack's bottom-most valid word. At all times, the stack pointer must point to a 0 mod 16 aligned area. The stack pointer is also used to access any memory arguments upon entry to a function. Except in the case of dynamic stack allocation (e.g., alloca), this register is preserved across any functions called by the current function. A call to a function that does not preserve the stack pointer must notify the compiler, to cause the generation of code that behaves properly. Failure to notify the compiler leads to undefined behavior. The standard function calling sequence does not include any method to detect such failures. This allows the compiler to use the stack pointer to reference stack items without having to set up a frame pointer for this purpose.

- **r13** is reserved for use as a *thread pointer*. The usage of this register is ABI specific. Programs conforming to these conventions may not modify this register.

- **r32–r39 (in0–in7)** are used as incoming argument registers. Arguments beyond these registers appear in memory, as explained in Chapter 8. Refer to the discussion below on structures and unions.
- **r32–r127** are stacked registers. Code may allocate a register stack frame of up to 96 registers with the alloc instruction, and partition this frame into three regions: input registers (in0, in1, ...), local registers (loc0, loc1, ...), and output registers (out0, out1, ...). The input and local regions are automatic, and the output region is scratch. See Chapter 6 for more information.

5.3 Floating-point registers

Floating-point registers are used for floating-point computations and certain integer computations, such as multiply and divide. Table 5–2 lists the floating-point registers.

Table 5–2. Floating-Point Registers

Register	*Class*	*Usage*
f0	constant	Always 0.0
f1	constant	Always 1.0
f2–f5	preserved	
f6–f7	scratch	
f8–f15	scratch	Argument/return registers
f16–f31	preserved	
f32–f127	scratch	Rotating registers or scratch

- **f2–f5** and **f16–f31** are preserved floating-point registers, and can be used for any value that needs to be preserved across a procedure call. A procedure using one of the preserved floating-point registers must save and restore the caller's original contents without generating a NaT consumption fault. This can be done by using the stf.spill and ldf.fill instructions.
- **f8–f15** are used as incoming floating-point argument registers. Floating-point arguments are placed in these registers when possible. Arguments beyond the registers appear in memory, as explained in Section 8.5. Within the called function, these are local scratch registers and are not preserved for the caller.

  Floating-point return values also appear in these registers. Single, double, and extended values are all returned using the appropriate format.
- **f32–f127** can be used as rotating registers. They are available as normal scratch registers if rotation is not being used.

5.4 Predicate registers

Predicate registers are single-bit-wide registers used for controlling the execution of predicated instructions. Table 5–3 lists the predicate registers.

Table 5–3. Predicate Registers

Register	*Class*	*Usage*
p0	constant	always 1
p1–p5	preserved	fixed
p6–p15	scratch	fixed
p16–p63	preserved	rotating

5.5 Branch registers

Branch registers are used for making indirect branches. Table 5–4 lists the branch registers.

Table 5–4. Branch Registers

Register	*Class*	*Usage*
b0	scratch	Return link
b1–b5	preserved	
b6–b7	scratch	

- **b0** contains the return address on entry to a procedure; it is a scratch register otherwise.

5.6 Application registers

Application registers are special-purpose registers designated for application use. Table 5–5 lists the application registers.

Table 5–5. Application Registers

Register	*Class*	*Usage*
ar.fpsr	see below	Floating-point status register
ar.rnat	automatic	RSE NaT collection register
ar.unat	preserved	User NaT collection register
ar.pfs	special	Previous function state
ar.bsp	read-only	Backing store pointer
ar.bspstore	special	Backing store store pointer
ar.rsc	see below	RSE control
ar.lc	preserved	Loop counter
ar.ec	automatic	Epilog counter (preserved in ar.pfs)
ar.ccv	scratch	Compare and Exchange comparison value
ar.itc	read-only	Interval time counter
ar.k0–ar.k7	read-only	Kernel registers

- **ar.fpsr** is the floating-point status register. This register is divided into several fields:

  Trap Disable Bits (bits 5–0). The trap disable bits must be preserved by the callee, except for procedures whose documented purpose is to change these bits.

  Status Field 0. The control bits must be preserved by the callee; except for procedures whose documented purpose is to change these bits. The flag bits are the IEEE floating point standard sticky bits and are part of the static state of the machine.

  Status Field 1. This status field is dedicated for use by divide and square root code, and must always be set to standard values at any procedure call boundary (including entry to exception handlers). These standard values are: trap disable set, round-to-nearest mode, 80-bit (extended) precision, widest range for exponent on, and flush-to-zero mode off. The flag bits are scratch.

  Status Fields 2 and 3. The control bits in these status fields must agree with the control bits in status field 0, and the trap disable bits should always be set at procedure calls and returns. The flag bits are always available for scratch use.

- **ar.rnat** holds the NaT bits for values stored by the register stack engine. These bits are saved automatically in the register stack backing store.

- **ar.unat** holds the NaT bits for values stored by the st8.spill instruction. As a preserved register, it must be saved before a procedure can issue any st8.spill instructions. The saved copy of ar.unat in a procedure's frame hold the NaT bits from the registers spilled by its caller; these NaT bits are thus associated with values local to the caller's caller.

- **ar.pfs** contains information that records the state of the caller's register stack frame and epilog counter. It is overwritten on a procedure call; therefore, it must be saved before issuing any procedure calls, and restored prior to returning.

- **ar.bsp** contains the address in the backing store corresponding to the base of the current frame. This register may be modified only as a side effect of writing ar.bspstore while the Register Stack Engine (RSE) is in enforced lazy mode.

- **ar.bspstore** contains the address of the next RSE store operation. It may be read or written only while the RSE is in enforced lazy mode. Under normal operation, this register is managed by the RSE, and application code should not write to it, except when performing a stack switching operation.

- **ar.rsc** is the register stack configuration register. This register is divided into several fields:

  Mode. This field controls the RSE behavior, and has scratch behavior. On a return, this field may be set to a standard value.

  Privilege level. This field controls the privilege level at which the RSE operates, and may not be changed by non-privileged software.

  Endian mode. This field controls the byte ordering used by the RSE, and should not be changed by an application.

Load distance (loadrs). This field is used by the `loadrs` instruction, and has scratch behavior.

Chapter 6

Register Stack

General registers 32 through 127 form a register stack that is automatically managed across procedure calls and returns. Each procedure frame on the register stack is divided into two dyamically-sized regions—one for input parameters and local variables, and one for output parameters. On a procedure call, the registers are automatically renamed by the hardware so that the caller's output registers form the base of the callee's new register stack frame. On return, the registers are restored to the previous state, so that the input and local registers are preserved across the call.

The alloc instruction is used at the beginning of a procedure to allocate the input, local, and output regions; the sizes of these regions are supplied as immediate operands. A procedure is not required to issue an alloc instruction if it does not need to store any values in its register stack frame. It may still read values from input registers, but it may not write to a stack register without first issuing an alloc instruction.

Figure 6–1 illustrates the operation of the register stack across an example procedure call. In this example, the caller allocates eight input, twelve local, and four output registers, and the callee allocates four input, six local, and five output registers.

The actual registers to which the stacking registers are physically mapped are not directly addressable by the application software.

6.1 Input and local registers

The hardware makes no distinction between input and local registers. The caller's output registers automatically become the callee's entire register stack frame on a procedure call, with all registers initially allocated as output registers. An alloc instruction may increase or decrease the total size of the register stack frame, and may adjust the boundary between the input and local region and the output region.

The software conventions specify that up to eight registers are used for parameter passing. Any registers in the input and local region beyond those eight may be allocated for use as preserved locals. Floating-point parameters may produce "holes" in the parameter list that is passed in the general registers; those unused input registers may also be used for preserved locals.

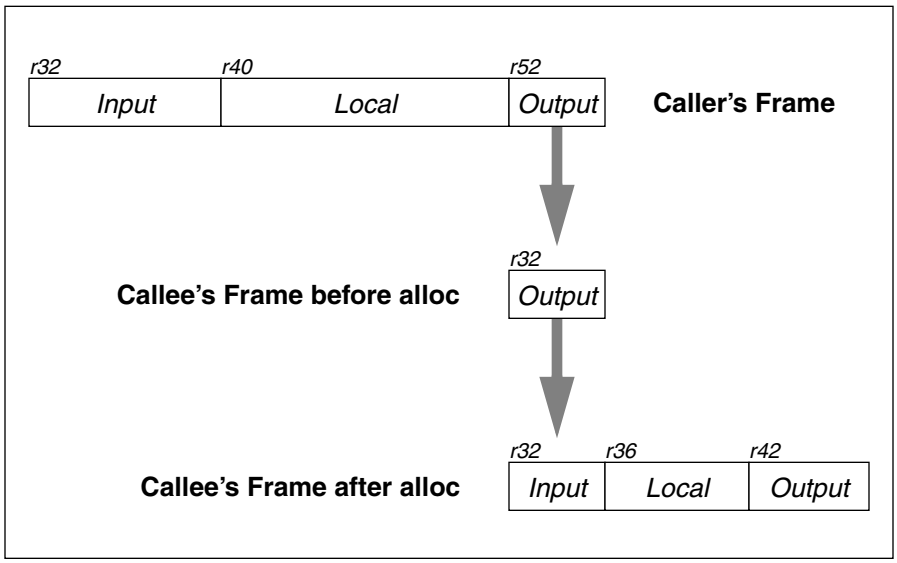


Figure 6–1. Operation of the Register Stack

The caller's output registers do not need to be preserved for the caller. Once an input parameter is no longer needed, or has been copied elsewhere, that register may be reused for any other purpose within the procedure.

6.2 Output registers

Up to eight output registers are used for passing parameters. If a procedure call requires fewer than eight general registers for its parameters, the calling procedure does not need to allocate more than are needed. If the called procedure expects more parameters, it will allocate extra input registers; these registers will be uninitialized.

A procedure may also allocate more than eight registers in the output region. While the extra registers may not be used for passing parameters, they can be used as extra scratch registers. On a procedure call, they will show up in the called procedure's output area as excess registers, and may be destroyed by that procedure. The called procedure may also allocate few enough total registers in its stack frame that the top of the called procedure's frame is lower than the caller's top of frame, but those registers will become available again when control returns to the caller.

6.3 Rotating registers

A subset of the registers in the procedure frame may be designated as rotating registers. The rotating register region always starts with r32, and may be any multiple of eight registers in number, up to a maximum of 96 rotating registers. The renaming is under control of the Rotating Register Base (RRB).

If the rotating registers include any or all of the output registers, software must be careful when using the output registers for passing parameters, since a non-zero RRB will change the virtual register numbers that are part of the output region. In general, software should either ensure that the rotating region does not overlap the output region, or that the RRB is cleared to zero before setting output parameter registers.

6.4 Frame markers

The current application-visible state of the stack frame is stored in an architecturally inaccessible register called the current frame marker. On a procedure call, this register is automatically saved by copying it to an application register, the previous function state (ar.pfs). The current frame marker is modified to describe a new stack frame whose input and local area is initially zero size, and whose output area is equal in size to the previous output area. On return, the previous frame state register is used to restore the current frame marker to its earlier value, and the base of the register stack is adjusted accordingly.

It is the responsibility of a procedure to save the previous function state register before issuing any procedure calls of its own, and to restore it before returning.

6.5 Backing store for register stack

When the depth of the procedure call stack exceeds the capacity of the physical register file, the hardware frees physical registers by saving them into a memory stack. This backing store is distinct from the memory stack described in the next chapter.

As returns unwind the procedure call stack, the hardware also restores previously-saved physical registers from the backing store.

The operation of this register stack engine (RSE) is mostly transparent to application software. While the RSE is running, application software may not examine the contents of the backing store, and may not make any assumptions about how much of the register stack is still in physical registers or in the backing store. In order to examine previous stack frames, application software must synchronize the RSE with the flushrs instruction. Synchronizing the RSE forces all stack frames up to, but not including, the current frame to be saved in backing store, allowing the software to examine the contents of the backing store without asynchronous operations modifying the memory. Modifications to the backing store require setting the RSE to “enforced lazy mode” after synchronizing it, which prevents the RSE from doing any operations other than those required by calls and returns. The procedure for synchronizing the RSE and setting the mode is described in Section 10.2.

The backing store grows towards higher addresses. When the RSE is synchronized and in enforced lazy mode, the top of the stack corresponding to the top of the previous procedure frame is available in the Backing Store Pointer (bsp) application register.

Even when the RSE is in enforced lazy mode, the bsp must always point to a valid backing store address, since the operating system may need to start the RSE to process an exception.

A NaT collection register is stored into the backing store after each group of 63 physical registers. For each register stored, its NaT bit is shifted into the collection register. When the bsp reaches the doubleword just before a 64 doubleword boundary, the RSE stores the collection register. Software can determine the position of the NaT collection registers in the backing store by examining the memory address. This process is described in greater detail in the EAS.

Chapter 7

Memory Stack

The memory stack is used for local dynamic storage, spilled registers, and parameter passing. It is organized as a stack of *procedure frames*, beginning with the main program's frame at the base of the stack, and continuing towards the top of the stack with nested procedure calls. At the top of the stack is the frame for the currently active procedure. (There may be some system-dependent frames at the base of the stack, prior to the main program's frame, but an application program may not make any assumptions about them.)

The memory stack begins at an address determined by the operating system, and grows towards lower addresses in memory. The stack pointer register, `sp`, always points to the lowest address in the current, top-most, frame on the stack.

Each procedure creates its frame on entry by subtracting its frame size from the stack pointer, and removes its frame from the stack on exit by restoring the previous value of `sp` (usually by adding its frame size, but a procedure may save the original value of `sp` when its frame size may vary).

Because the register stack is also used for the same purposes, not all procedures will need a stack frame. Every non-leaf procedure, however, needs to save at least its return link and the previous frame marker either on the register stack or in the memory stack, so there is an activation record for every non-leaf procedure on one or both of the stacks.

7.1 Procedure frames

A procedure frame consists of five regions, as illustrated in Figure 7–1.

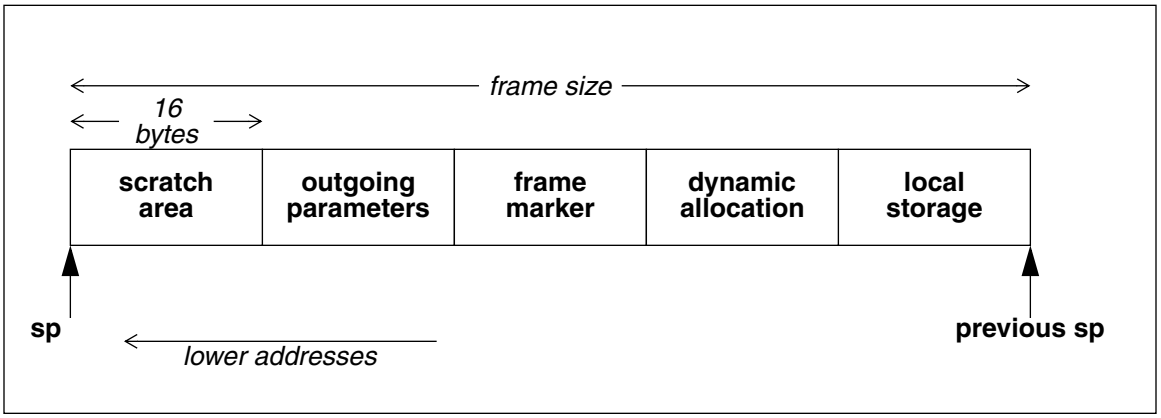


Figure 7–1. Procedure frame

These regions are:

- Local storage. A procedure may store local variables, temporaries, and spilled registers in this region.
- Dynamically-allocated stack storage. This is a variable-sized region (initially zero length), that can be created by the C library `alloca` routine and similar routines.
- Frame marker. This optional region may contain information required for unwinding through the stack (for example, a copy of the previous stack pointer).
- Outgoing parameters. Parameters in excess of those passed in registers are stored in this region of the stack frame. A procedure accesses its incoming parameters in the outgoing parameter region of its caller's stack frame.
- Scratch area. This 16-byte region is provided as scratch storage for procedures that are called by the current procedure. Leaf procedures do not need to allocate this region. A procedure may use the 16 bytes at the top of its own frame as scratch memory, but the contents of this area may be destroyed by a procedure call.

The stack pointer must always be aligned at a 16-byte boundary. This implies that all stack frames must be a multiple of 16 bytes in size.

An application may not write to memory below the stack pointer. Any memory below the stack pointer may be destroyed at any time.

Most procedures are expected to have a fixed size frame, and the conventions are biased in favor of this. A procedure with a fixed size frame may reference all regions of the frame with a compile-time constant offset relative to the stack pointer. Compilers should determine the total size required for each region, and pad the local storage area to make the total frame size a multiple of 16 bytes. The procedure may then create the frame by subtracting an

immediate constant from the stack pointer in the prologue, and remove the frame by adding the same immediate to the stack pointer in the epilogue.

If a procedure has a variable-size frame (for example, it contains a call to `alloca`), it should make a copy of `sp` to serve as a frame pointer before subtracting the initial frame size from the stack pointer. It may then restore the previous value of the stack pointer in the epilogue without regard for how much dynamic storage has been allocated within the frame. It may also use the frame pointer to access the local storage region, since offsets from `sp` will vary.

A frame pointer, as described above, is not required, however, provided that the compiler uses an equivalent method of addressing the local storage region correctly before and after dynamic allocation, and provided that the code satisfies conditions imposed by the stack unwind mechanism.

To expand a stack frame dynamically, the scratch area, outgoing parameters, and frame marker regions, which are always located relative to the current stack pointer must be relocated to the new top of stack. If the scratch area and outgoing parameter area are both clear of any live values, there is no actual work involved in relocating these areas. For procedures with dynamically-sized frames, it is recommended that the previous stack pointer value be stored in a local stacked general register instead of the frame marker, so that the frame marker is also empty. If the previous stack pointer is stored in the frame marker, the code must take care to ensure that the stack is always unwindable while the stack is being expanded (see Chapter 11).

Other issues depend on the compiler and the code being compiled. The standard calling sequence does not define a maximum stack frame size, nor does it restrict how a language system uses any stack frame region beyond those purposes described here. For example, the outgoing parameter region may be used as scratch storage whenever it is not needed for passing parameters.

Chapter 8

Procedure Linkage

This chapter discusses linkage conventions and the details of the procedure calling sequence, including parameter passing and return values.

8.1 External naming conventions

The standard naming convention, referred to as the "C" convention, specifies that all external symbols have linkage names identical to the source language identifier. There are no leading or trailing underscores. Other languages may establish other conventions, but they should provide a mechanism to define and reference symbols with "C" linkage.

8.2 The gp register

Every procedure that references statically-allocated data or calls another procedure requires a pointer to its data segment in the `gp` register, so that it can access its static data and its linkage tables. Each load module has its own data segment, and the `gp` register must be set correctly prior to calling any entry point within that load module.

The linkage conventions require that each load module define exactly one `gp` value to refer to a location within its short data segment. It is expected that this location will be chosen to maximize the usefulness of short-displacement immediate instructions for addressing scalars and linkage table entries. The DLL loader will determine the absolute value of the `gp` register for each load module after loading its data segment into memory.

For calls within a load module, the `gp` register will remain unchanged, so calls known to be local can be optimized accordingly.

For calls between load modules, the `gp` register must be initialized with the correct `gp` value for the new load module, and the calling function must ensure that its own `gp` value is saved and restored.

8.3 Types of calls

The following types of procedure calls are defined:

- Direct calls. Direct calls within the same load module may be made directly to the entry point of the target procedure. In this case, the gp register does not need to be changed.
- Direct dynamically-linked calls. These calls are routed through an import stub (which may be inlined at compile time if the call is known or suspected to be to another load module). The import stub obtains the address of the main entry point and the gp register value from the linkage table. Although coded in source as a direct call, dynamically-linked calls become indirect.
- Indirect calls. A function pointer must point to a descriptor that contains both the address of the function entry point and the gp register value for the target function. The compiler must generate code for an indirect call that sets the new gp value before transferring control to the target procedure.
- Special calls. Other special calling conventions are allowed to the extent that the compiler and the runtime library agree on convention, and provided that the stack may be unwound through such a call. Such calls are outside the scope of this document. See Section 8.7 for a discussion of stack unwind requirements.

8.4 Calling sequence

Direct and indirect procedure calls are described in the following sections. Since the compiler is not required to know whether any given call is local or to another load module, the two types of direct calls are described together in the first section.

8.4.1 Direct calls

Direct procedure calls follow the sequence of steps shown in Figure 8–1. The following paragraphs describe these steps in detail.

Preparation for call. Values in scratch registers that must be kept live across the call must be saved. They can be saved by copying them into local dynamic registers, or by saving them on the memory stack. If the NaT bits associated with any live scratch registers must be saved, the compiler should use st8.spill or stf.spill instructions. The User NaT collection register itself is preserved by the call, so the NaT bits need no further treatment at this point.

If the call is not known (at compile time) to be within the same load module, the gp register must be saved.

The parameters must be set up in registers and memory as described in Section 8.5.

Procedure call. All direct calls are made with a br.call instruction, specifying BR 0 (also known as rp) for the return link.

For direct local calls, the pc-relative displacement to the target is computed at link time. Compilers may assume that the standard displacement field in the

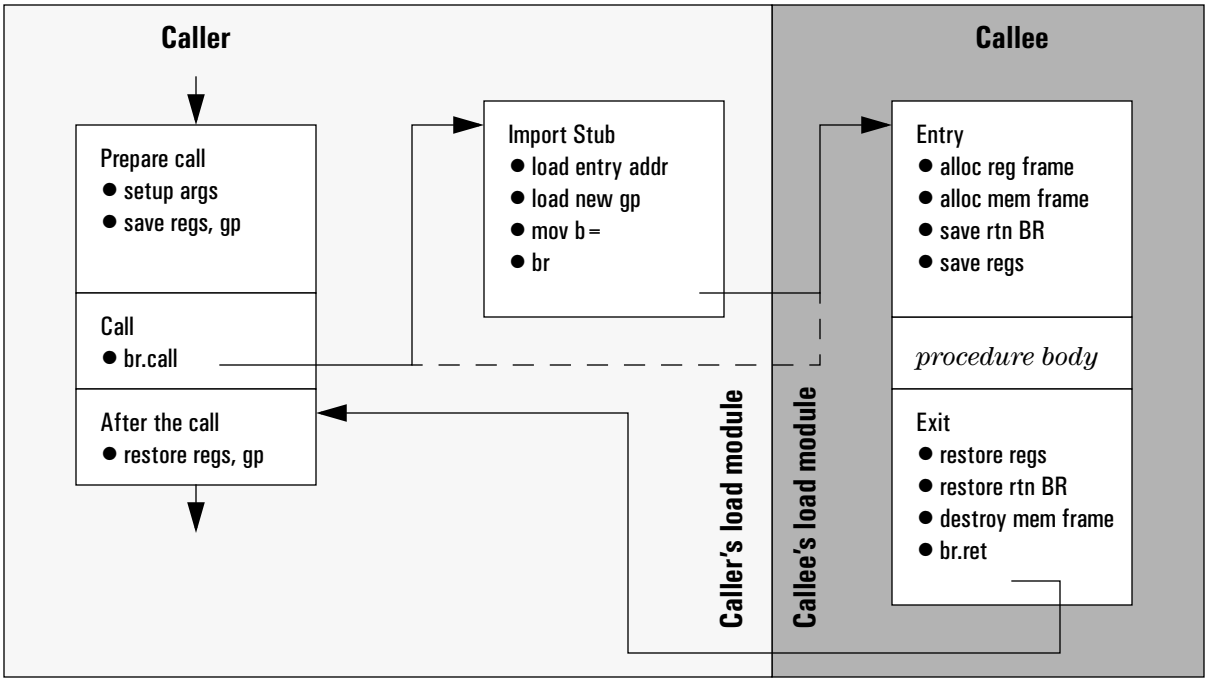


Figure 8–1. Direct procedure calls

br.call instruction is sufficiently wide to reach the target of the call. If the displacement is too large, the linker must supply a branch stub at some convenient point in the code; compilers must guarantee the existence of such a point by ensuring that code sections in the relocatable object files are no larger than the maximum reach of the br.call instruction. With a 25-bit displacement, the maximum reach is 16 megabytes in either direction from the point of call.

Direct calls to other load modules cannot be statically bound at link time, so the linker must supply an import stub for the target procedure; the import stub obtains the address of the target procedure from the linkage table. The br.call instruction can then be statically bound using the pc-relative displacement to the import stub.

The br.call instruction saves the return link in the return BR, saves the current frame marker in the ar.pfs register, and sets the base of the new register stack frame to the beginning of the output region of the old frame.

Import stub (direct external calls only). The import stub is allocated in the load module of the caller, so that the br.call instruction may be statically bound to the address of the import stub. It must access the linkage table via the current gp (which means that gp must be valid at the point of call), and obtain the address of the target procedure's entry point and its gp value. The import stub then establishes the new gp value and branches to the target entry point.

If the compiler knows or suspects that the target of a call is in a separate load module, it may wish to generate calling code that performs the functions of the import stub, saving an extra branch. The detailed operation of an import stub, however, is ABI specific.

When the target of a call is in the same load module, an import stub is not used (which also means that gp must be valid at the point of call).

Procedure entry. The prologue code in the target procedure is responsible for allocating the register stack frame, and a frame on the memory stack, if necessary. It may use the 16 bytes at the top of its caller's memory stack frame as scratch area.

If it is a non-leaf procedure, it must save the return BR and previous frame marker, either in the memory stack frame or in a local dynamic GR.

The prologue must also save any preserved registers that will be used in this procedure. The NaT bits for those registers must be preserved as well, by copying to local stacked general registers, or by using st8.spill or stf.spill instructions. The User NaT collection register (ar.unat) must be saved first, however, since it is guaranteed to be preserved by the call.

Procedure exit. The epilogue code is responsible for restoring the return BR and previous frame marker, if necessary, and any preserved registers that were saved. The NaT bits must be restored using the ld8.fill or ldf.fill instructions. The User NaT collection register must also be restored if it was saved.

If a memory stack frame was allocated, the epilogue code must deallocate it.

Finally, the procedure exits by branching through the return BR with the br.ret instruction.

After the call. Any saved values (including gp) should be restored.

8.4.2 Indirect calls

Indirect procedure calls follow nearly the same sequence, except that the branch target is established indirectly. This sequence is illustrated in Figure 8–2.

Function Pointers. A function pointer is always the address of a function descriptor for the target procedure. The function descriptor must be allocated in the data segment of the target procedure, because it contains pointers that must be relocated by the DLL loader.

The function descriptor contains at least two 64-bit double-words: the first is the entry point address, and the second is the gp value for the target procedure. An indirect call will load the gp value into the gp register before branching to the entry point address.

In order to guarantee the uniqueness of a function pointer, and because its value is determined at program invocation time, code must materialize function pointers only by loading a pointer from the data segment. The object file format will provide appropriate relocations for this pointer.

Preparation for call. Indirect calls are made by first loading the function pointer into a general register, loading the entry point address and the new gp

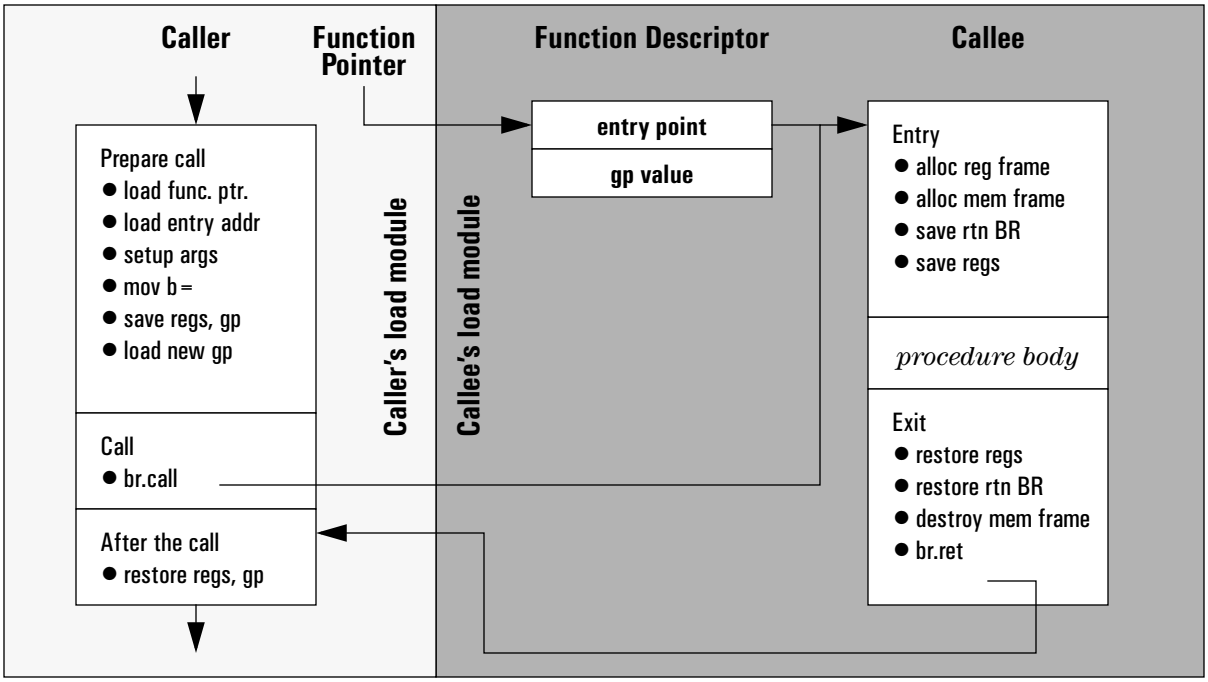


Figure 8–2. Indirect Procedure Calls

value, then using the Move to Branch Register operation to move the address of the procedure entry point into the BR to be used for the call.

Values in scratch registers that must be kept live across the call must be saved. They can be saved by copying them into local dynamic registers, or by saving them on the memory stack. If the NaT bits associated with any live scratch registers must be saved, the compiler should use st8.spill or stf.spill instructions. The User NaT collection register itself is preserved by the call, so the NaT bits need no further treatment at this point.

Unless the call is known (at compile time) to be within the same load module, the gp register must be saved before the new gp value is loaded.

The parameters must be set up in registers and memory as described in Section 8.5.

Procedure call. All indirect calls are made with the indirect form of the br.call instruction, specifying BR 0 (also known as rp) for the return link.

The br.call instruction saves the return link in the return BR, saves the current frame marker in the ar.pfs register, and sets the base of the new register stack frame to the beginning of the output region of the old frame. Because the indirect call sequence obtains the entry point address and new gp value from the function descriptor, control flows directly to the target procedure, without the need for any intervening stubs.

Procedure entry, exit, and return. The remainder of the calling sequence is the same as for direct calls.

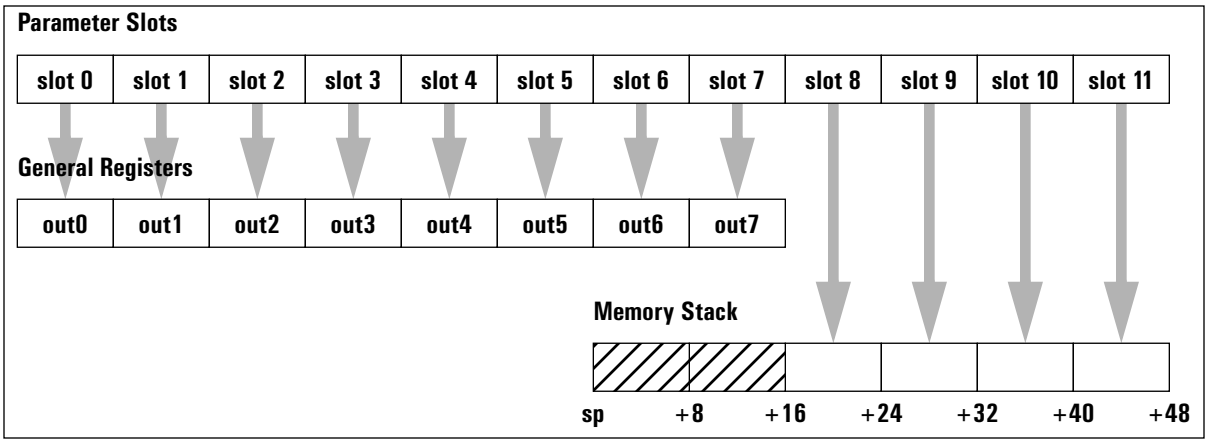


Figure 8–3. Parameter Passing in General Registers and Memory

8.5 Parameter passing

Parameters are passed in a combination of general registers, floating-point registers, and memory, as described below, and as illustrated in Figure 8–3.

The parameter list is formed by placing each individual parameter into fixed-size elements of the parameter list, referred to as parameter slots. Each parameter slot is 64 bits wide; parameters larger than 64 bits are placed in as many consecutive parameter slots as are needed to contain the entire parameter. The rules for allocation and alignment of parameter slots are given later in this section.

The contents of the first eight parameter slots are always passed in registers, while the remaining parameters are always passed on the memory stack, beginning at the caller's stack pointer plus 16 bytes. The caller uses up to eight of the registers in the output region of its register stack for integer parameters, and up to eight floating-point registers for floating-point parameters.

To accomodate variable argument lists in the C language, there is a fixed correspondence between parameter slots and output registers used for general register arguments. This allows a procedure to spill its register parameters easily to memory before stepping through the parameter list with a pointer. Also because of variable argument lists, floating-point parameters are sometimes passed in both general output registers and in floating-point registers.

There is no fixed correspondence between parameter slots and floating-point parameter registers. Parameters passed in floating-point registers always use the next available floating-point parameter register, starting with f8.

A procedure may assume that the NaT bits on its incoming general register arguments are clear, and that the incoming floating-point register arguments are not NaTVals. A procedure making a call must ensure only that registers containing actual parameters are clear of NaT bits or NaTVals; registers not used for actual parameters may contain garbage. (See the rationale in Section .)

Allocation of Parameter Slots

Parameters slots are allocated for each parameter, based on the parameter type and size, treating each parameter in sequence, from left to right. The rules for allocating parameter slots and placing the contents within the slot are given in Table 8–1.

Table 8–1. Rules for Allocating Parameter Slots

Type	*Size (Bits)*	*Allocation*	*Number of Slots*	*Alignment*
Integer/Pointer	1–64	Next Available	1	LSB
Integer	65–128	Next Even	2	LSB
Single-Precision Floating-Point	32	Next Available	1	LSB
Double-Precision Floating-Point	64	Next Available	1	LSB
Double-Extended Floating-Point	80	Next Even	2	Byte 0
Quad-Precision Floating-Point	128	Next Even	2	Byte 0
Aggregates	any	Next Aligned	(size+63)/64	Byte 0

Note *These rules are applied based on the type of the parameter after any type promotion rules specified by the language have been applied. For example, a short integer passed without a function prototype in C would be promoted to the int type, and would be passed according to the rules for the int type.*

The allocation column of the table indicates how parameter slots are allocated for each type of parameter.

- "Next Available" means that the parameter is placed in the slot immediately following the last slot used.
- "Next Even" means that the parameter is placed in the next available even-numbered slot, skipping an odd-numbered slot if necessary. If an odd-numbered slot is skipped, it will not be used for any subsequent parameters.
- "Next Aligned" means that the allocation is dependent on the external alignment of the aggregate; that is, on the alignment boundary required for the aggregate as a whole. For aggregates with an external alignment of 1–8 bytes, the "Next Available" policy is used; for aggregates with an external alignment of 16 bytes, the "Next Even" policy is used.

This placement policy ensures that parameters will fall on a natural alignment boundary if passed in memory.

The alignment column of the table indicates how parameters are aligned within a parameter slot. There are two kinds of alignment, "LSB" and "Byte 0."

Byte Order

- "LSB" alignment specifies that the least-significant bit of the parameter is aligned with the least-significant bit of the argument slot or slots (i.e., right aligned). Parameters shorter than 64 or 128 bits are padded on the left; the padding is undefined. When a pair of parameter slots is required, the even-numbered parameter slot contains the most-significant bits in big-endian environments, and the least-significant bits in little-endian environments. See Figure 8–4 for examples.

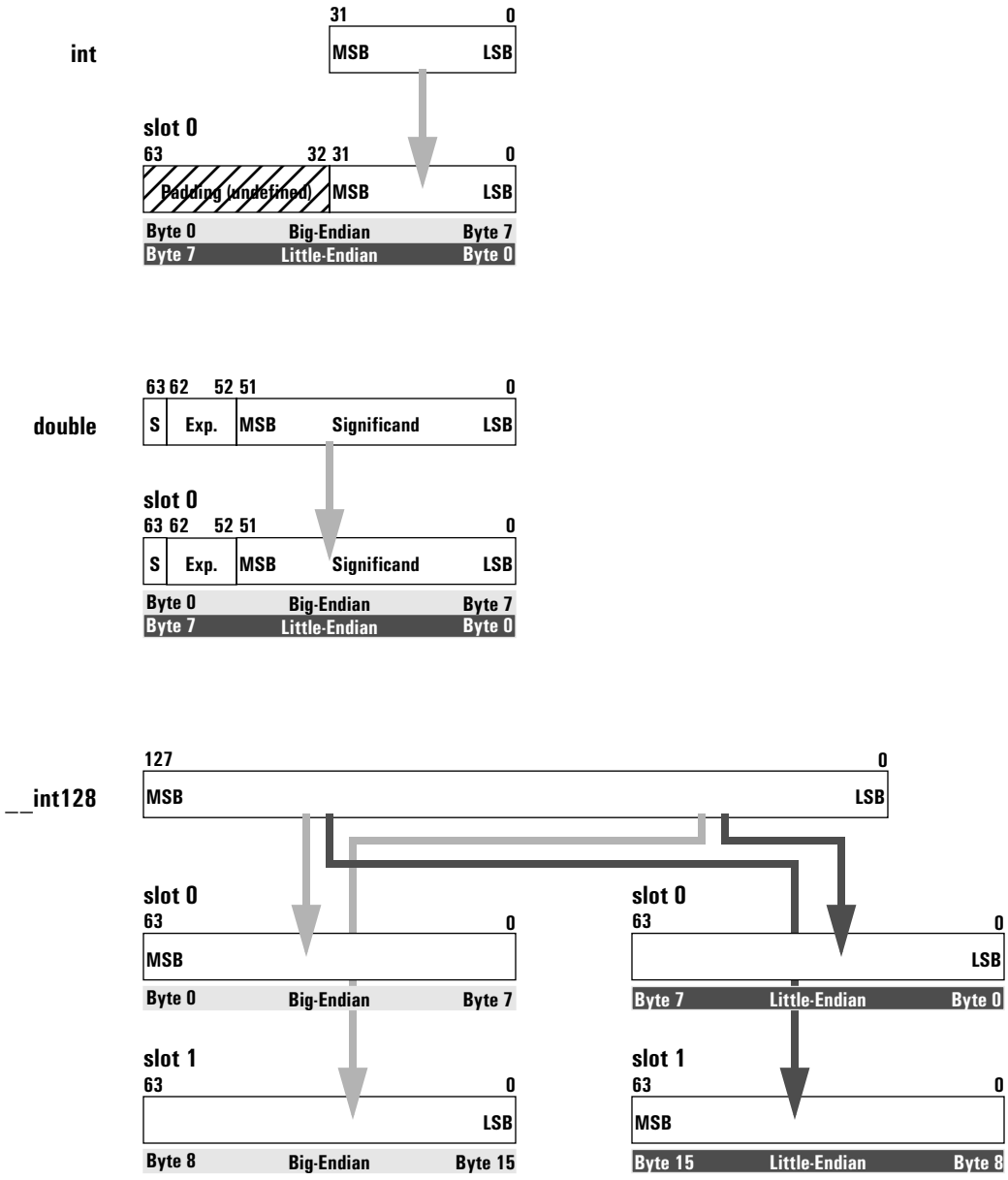


Figure 8–4. Examples of "LSB" Alignment

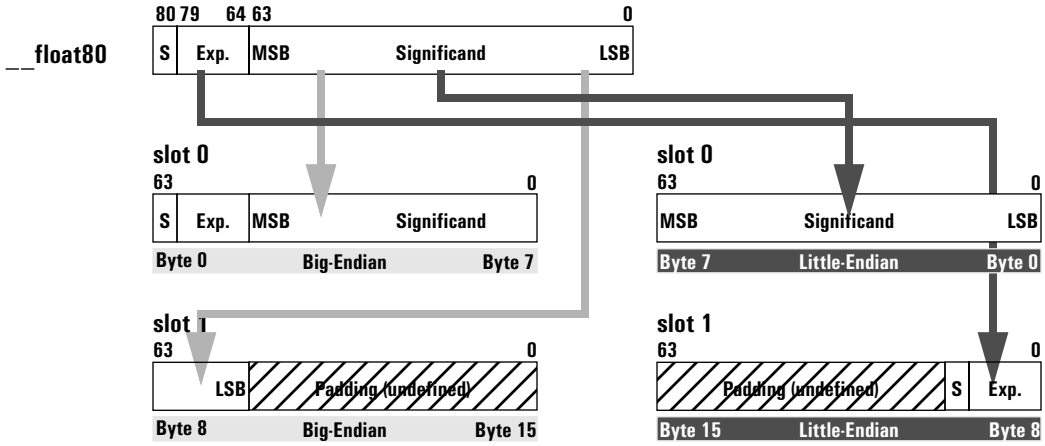


Figure 8–5. Example of "Byte 0" Alignment

Byte Order

- "Byte 0" alignment specifies that byte 0 of the parameter is aligned with byte 0 of the parameter slot. Parameters that are not a multiple of 64 bits in length are padded at the end; the padding is undefined. In big-endian environments, the padding will be at the right end of the final parameter slot; in little-endian environments, the padding will be at the left end of the final parameter slot. See Figure 8–5 for an example.

Register parameters

The first eight parameter slots (64 bytes) are passed in registers, according to the rules in this section.

- These eight argument slots are associated, one-to-one, with the stacked output GRs, as shown in Figure 8–3.
- Integral scalar parameters, quad-precision (128-bit) floating-point parameters, and aggregate parameters in these slots are passed only in the corresponding output GRs. Aggregates consisting solely of floats, of doubles, or of double-extended values are an exception; see below.
- If an aggregate parameter straddles the boundary between slot 7 and slot 8, the part that lies within the first eight slots is passed in GRs, and the remainder is passed in memory, as described in the next section.

Single-precision, double-precision, and double-extended-precision floating-point scalar parameters in these slots are passed according to the available formal parameter information at the point of call (for example, from a function prototype).

If an actual parameter is known to correspond to a floating-point formal parameter, the following rules apply:

- The actual parameter is passed in the next available floating-point parameter register, if one is available. Floating-point parameter registers are allocated as needed from the range f8-f15, starting with f8.

- If all available floating-point parameter registers have been used, the actual parameter is passed in the appropriate general register(s). (This case can occur only as a result of homogeneous floating-point aggregates, described below.)

If a floating-point actual parameter is known to correspond to a variable-argument specification in the formal parameter list, the following rule applies:

- The actual parameter is passed in the appropriate general register(s).

If the compiler cannot determine, at the point of call, whether the corresponding formal parameter is a varargs parameter, it must generate code that satisfies both of the above conditions. (The compiler's determination may be based on prototype declarations, language standard assumptions, analysis, or other user options or information.)

When floating-point parameters are passed in floating-point registers, they are passed in the register format, rounded to the appropriate precision. When passed in general registers, floating-point values are passed in their memory format.

Parameters allocated beyond the eighth parameter slot are never passed in registers, even when floating-point parameter registers remain unused.

Aggregates whose elements are all single-precision, all double-precision, or all double-extended-precision values (but not quad-precision), are treated specially. These "homogeneous floating-point aggregates" (HFAs) may be arrays of one of these types, structures whose only members are all one of these types, or structures that contain other structures, provided that all lowest-level members are one of these types, and all are the same type. (This definition includes Fortran COMPLEX data, except COMPLEX*32.) The following additional rules apply to these types of parameters (but only to the portion of an aggregate that lies within the first eight argument slots):

The following rules apply to floating-point aggregates:

- If an actual parameter is known to correspond to an HFA formal parameter, each element is passed in the next available floating-point argument register, until the eight argument registers are exhausted. The remaining elements of the aggregate are passed in output GRs, according to the normal conventions.
- If an actual parameter is known to correspond to a variable-argument specification, the aggregate is passed as any other aggregate.

If the compiler cannot determine, at the point of call, whether the corresponding formal parameter is a varargs parameter, the elements of the aggregate must be passed in both the corresponding output GRs and in floating-point argument registers.

Note *Because HFAs are mapped to parameter slots as aggregates, single-precision HFAs will be allocated with two floating-point values in each parameter slot, but only one value per register. Thus, the available floating-point parameter registers may become exhausted before the end of the first eight parameter slots, and additional members of the HFA must be passed in general registers.*

It is possible for the first of two values in a parameter slot to occupy the last available floating-point parameter register. In this case, the second value is passed in its designated GR, but the half of the GR that would have contained the first value is undefined.

Memory stack parameters

The remainder of the parameter list, beginning with slot 8, is passed in the outgoing parameter area of the memory stack frame, as described in Section 7.1. Parameters are mapped directly to memory, with slot 8 placed at location `sp+16`, slot 9 at `sp+24`, and so on. Each argument slot is stored in memory as a 64-bit storage unit according to the byte order of the current environment.

Variable argument lists

The rules above support variable-argument list functions in both the K&R and the ANSI C languages. When an ANSI prototype is in scope, all register parameters corresponding to a variable-argument specification are passed in GRs; when no prototype is in scope, floating-point parameters are copied in both GRs and FRs because of the possibility that the callee is expecting a variable argument list.

Thus, a function with variable arguments may assume that the variable arguments that lie within the first eight argument slots can all be found in the stacked input GRs, `in0`–`in7`. It may then store these registers to memory, using the 16-byte scratch area for `in6` and `in7`, and using up to 48 bytes at the base of its own stack frame for `in0`–`in5`, as necessary. This arrangement places all the variable parameters in one contiguous block of memory.

When storing registers to memory for this purpose, the code must use the `st8.spill` instruction, since the registers are not guaranteed to contain valid values.

Byte Order

In a big-endian environment, the alignment and padding rules require the code that steps through the argument list to distinguish between aggregates and integers smaller than 8 bytes. Aggregates will be left-aligned within an 8-byte slot, while integers will be right-aligned.

Examples of the macros from the `<stdarg.h>` header file are given in Appendix A.

Pointers to formal parameters

Whenever the address is formed of a formal parameter that is passed in a register, the compiler must store the parameter to the stack, as it would for a variable argument list.

Languages other than C

Most languages other than C can usually be treated as if prototypes are always in scope, avoiding the need to pass floating-point parameters in both GRs and FRs. For example, because Fortran passes floating-point parameters by value only when calling an intrinsic function, it may safely assume that the callee is expecting the parameter in an FR.

A compiler for another language may need to honor the variable-argument list conventions, however, if it provides a mechanism for calling C procedures that may have variable-argument lists.

Rounding floating-point values

Floating-point parameters passed in floating-point registers should always be explicitly rounded to the proper precision expected by the language. There should be no difference in behavior between a floating-point parameter passed directly in registers and a floating-point parameter that has been stored to memory and reloaded.

Examples

The following examples illustrate the parameter passing conventions.

Example 8-1. Scalar integers and floats, with prototype

```
extern int func(int, double, double, int);
func(i, a, b, j);
```

The parameters are passed as follows:

i	out0
a	f8
b	f9
j	out3

Example 8-2. Scalar integers and floats, without prototype

```
extern int func();
func(i, a, b, j);
```

The parameters are passed as follows:

i	out0
a	out1 and f8
b	out2 and f9
j	out3

Example 8-3. Aggregates passed by value

```
extern int func();
struct { int array[20]; } a;
func(i, a);
```

The structure's external alignment is only 4 bytes, so no padding is required in the parameter list. The parameters are passed as follows:

i	out0
a.array[0–13]	out1–out7
a.array[14–19]	In memory, at sp+16 through sp+39

Example 8-4. Aggregates passed by value

```
extern int func();
struct { __float128 x; int array[20]; } a;
func(i, a);
```

The structure's external alignment is 16 bytes, so parameter slot 1 is skipped. The parameters are passed as follows:

i	out0
a.x	out2–out3
a.array[0–7]	out4–out7
a.array[8–19]	In memory, at sp+16 through sp+63

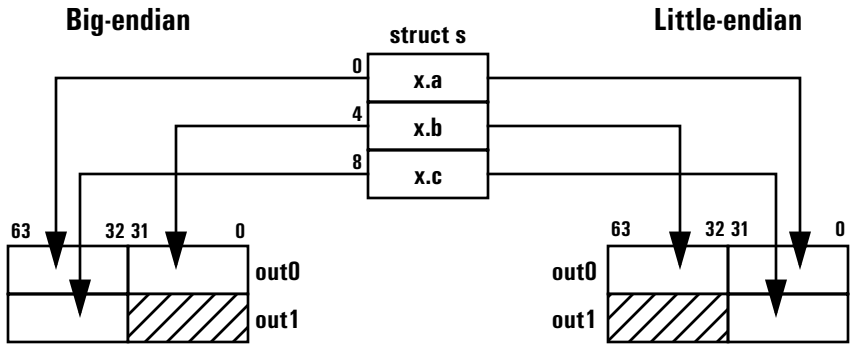
Example 8-5. Floating-point aggregates, without prototype

```
struct s {float a, b, c;} x;
extern func();
func(x);
```

The parameters are passed as follows:

x.a	out0 and f8
x.b	out0 and f9
x.c	out1 and f10

In little-endian environments, x.a and x.c are in the least-significant bits of out0 and out1, respectively, while x.b is in the most-significant bits of out0. In big-endian environments, x.a and x.c are in the most-significant bits of out0 and out1, respectively, while x.b is in the least-significant bits of out0.



Example 8-6. Floating-point aggregates, with prototype

```
struct s {float a, b, c;} x;
extern void func(struct s);
func(x);
```

The parameters are passed as follows:

```
x.a     f8
x.b     f9
x.c     f10
```

8.6 Return values

Values up to 256 bits and certain aggregates are returned directly in registers, according to the rules in Table 8–2.

Table 8–2. Rules for Return Values

Type	*Size (Bits)*	*Location of Return Value*	*Alignment*
Integer/Pointer	1–64	r8	LSB
Integer	65–128	r8, r9	LSB
Single-Precision Floating-Point	32	f8	N/A
Double-Precision Floating-Point	64	f8	N/A
Double-Extended Floating-Point	80	f8	N/A
Quad-Precision Floating-Point	128	r8, r9	Byte 0
Single-Precision HFA	32–256	f8–f15	N/A
Double-Precision HFA	64–512	f8–f15	N/A
Double-Extended HFA	128–1024	f8–f15	N/A
Aggregates	1–64	r8	Byte 0
Aggregates	65–256	r8–r11	Byte 0
Aggregates	> 256	Memory	Byte 0

Byte Order

When multiple registers are used to return a numeric value, the lowest-numbered register contains the most-significant bits in big-endian environments, and the least-significant bits in little-endian environments. When multiple registers are used to return an aggregate, the lowest-numbered register contains the first eight bytes of the aggregate. In big-endian environments, the padding will be at the right end of the final register used; in little-endian environments, the padding will be at the left end of the final register used.

Integral return values smaller than 32 bits must be zero-filled (if unsigned) or sign-extended (if signed) to at least 32 bits.

Homogeneous floating-point aggregates, as defined in Section 8.5, are returned in floating-point registers, provided the array or structure contains no more than eight individual values. The elements of the aggregate are placed in successive floating-point registers, beginning with f8. If the array or structure

contains more than eight elements, it is returned according to the rule below for aggregates larger than 256 bits.

Return values larger than 256 bits (except HFAs of up to 8 elements) are returned in a buffer allocated by the caller. A pointer to the buffer is passed to the called procedure in r8. This register is not guaranteed to be preserved by the called procedure (that is, the caller must preserve the address of the buffer through some other means). The return buffer must be aligned at a 16-byte boundary. A procedure may assume that the return buffer does not overlap any data that is visible to it through any other names.

A procedure may assume that any procedure it calls will return a valid value (i.e., the NaT bits are clear if the return is in general registers, and floating-point values returned are not NaTVals).

8.7 Requirements for unwinding the stack

Certain constraints must be met in order to unwind the stack successfully at any time, both by standard procedure calls as described here, and by special-purpose calling conventions. Chapter 11 describes how the unwind process works and the format of the unwind data structures. To meet the needs of the stack unwind mechanism, the following rules must be followed at all times:

- The previous function state register (ar.pfs) must be preserved prior to any call. The compiler must record, in the unwind data structures, where this register is stored, and over what range of code the saved value is valid.
- For special calls using a return BR other than b0, the compiler must record the BR number used for the return link.
- The return BR must be preserved prior to any call involving the same BR. The compiler must record where the return BR is stored and over what range of code the saved value is valid.
- If a procedure has a memory stack frame, the compiler must record either: (1) how large the frame is, or (2) that a previous frame pointer is stored on the stack or in a general register.

Chapter 9

Coding Conventions

This chapter discusses general coding conventions and presents some example code sequences for various tasks. The code sequences shown in this chapter are intended to serve as guidelines and examples rather than as required coding conventions. The requirements are documented in other chapters in this document.

9.1 Sample code sequences

In the sample code sequences in this section, registers of the form t1, t2, etc., are temporary registers, and may be assigned to any available scratch register. The code sequences show necessary cycle breaks, but no other scheduling considerations have been made. It is assumed that these code sequences will be scheduled with surrounding code to make best use of the processor resources.

9.1.1 Addressing “own” data in the short data area

“Own” short data may be addressed with a simple direct reference relative to the gp register, as illustrated below.

Example 9-1. Addressing “own” short data

```
addl        t1=@gprel(var),gp   // calc. address of var
;;
ld8         loc0=[t1]           // load contents of var
```

"Own" long data may be addressed either via the linkage table, as shown in Example 9-3, or directly as illustrated below.

Example 9-2. Addressing "own" long data

```
        movl    t1=@gprel(var)      // form gp-relative offset of var
        ;;
        add     t2=t1,gp            // calc. address of var
        ;;
        ld8     loc0=[t2]           // load contents of var
```

9.1.2 Addressing external data or data in a long data area

When data is not known to be defined in the current load module (i.e., it is not "own"), or if it is too large for the short data region, it must be accessed indirectly through the linkage table, as shown below.

Example 9-3. Addressing external or long data

```
        addl    t1=@ltoff(var),gp   // calc. address of LT entry
        ;;
        ld8     t2=[t1]             // load address of var
        ;;
        ld8     loc0=[t2]           // load contents of var
```

9.1.3 Addressing literals in the text segment

Literals in the text segment may be addressed either through the linkage table, as in Example 9-3 above, or with pc-relative addressing, as shown below. Note that the first two instructions may be moved towards the beginning of the procedure, and the base address of the literal area, in loc0, can be shared by other literal references in the same procedure.

Example 9-4. Addressing literals

```
L1:     mov     r3=ip               // get current IP
        ;;
        addl    loc0=litbase–L1,r3  // calc. addr. of lit. area
        ;;
        adds    t2=(lit–litbase),loc0 // calc. address of lit.
        ;;
        ld8     loc1=[t2]           // load value of literal
```

9.1.4 Materializing function pointers

Function pointers must always be obtained from the data segment, either as an initialized word or through the linkage table, as shown in the following examples:

Example 9-5. Materializing function pointers through linkage table

```
        addl    t1=@ltoff(@fptr(func)),gp // calc address of LT entry
        ;;
        ld8     loc0=[t1]         // load function pointer
```

Example 9-6. Materializing function pointers in data

```
fptr:   data8   @ftpr(func)       // initialize function ptr
```

9.1.5 Direct procedure calls

The following code sequence assumes that the parameters have already been placed in the proper locations.

Example 9-7. Direct procedure call

```
        mov     loc0=gp           // save current gp
        br.call rp=func           // make the call
        ;;
        mov     gp=loc0           // restore gp
```

9.1.6 Indirect procedure calls

The indirect procedure call sequence must load the function's entry point and gp value from the function descriptor. In this example, the function pointer is assumed to have been loaded into register loc0.

Example 9-8. Indirect Procedure Call

```
mov       loc1=gp          // save current gp
;;
ld8       t1=[loc0],8      // load entry point
;;
ld8       gp=[loc0]        // load new gp value
mov       b6=t1            // move ep to call BR
;;
br.call   rp=b6            // make the call
;;
mov       gp=loc1          // restore gp
```

9.1.7 Jump tables

High-level language constructs such as case and switch statements, where there are several possible local targets of a branch, may use a number of different code generation strategies, ranging from sequential conditional branches to a direct-lookup branch table.

If the compiler chooses to generate a branch table, the table should be placed in the text segment, and each table entry should be a 64-bit byte displacement from the base of the branch table to the branch target for that entry. This allows the displacements to be statically determined at link time, and no relocations will need to be applied at program invocation time. With displacements relative to the base address of the branch table, the code can easily add the displacement obtained from the table to the base address of the table to compute the target branch address.

A sample indirect branch is shown below. The branch table is assumed to be an array of 64-bit entries, each of which is an offset, relative to the beginning of the branch table, to the branch target. The branch table index is assumed to have been computed or loaded into register loc0.

Example 9-9. Branching Through a Branch Table

```
addl        loc1=@ltoff(brtab),gp // calc. address of
;;                                // linkage table entry

ld8         loc2=[loc1]           // load addr. of br. table
;;

shladd      loc3=loc0,3,loc2      // calc. address of branch
;;                                // table entry

ld8         loc4=[loc3]           // load branch table entry
;;

add         loc5=loc4,loc2        // calc. target address
;;

mov         b6=loc5               // move address to b6...
;;

br.cond     b6                    // ...and branch
;;
```

Alternatively, the code could use a pc-relative addressing sequence to obtain the base address of the jump table, using code similar to that in Example 9-4.

9.2 Speculation

Data speculation, using advanced load instructions, across procedure calls will not work correctly if the target of the advanced load is not one of the registers in the in/local region of the register stack frame. Upon return from the procedure call, the information in the ALAT could refer to an unchecked (or uncleared) advanced load to the same register from within the called procedure, rather than the information from the original load prior to the call.

Speculation recovery code may be placed within the procedure, outside the procedure but contiguous with it, or in a completely different section of memory. In any case, the target of the check instruction must be placed in or contiguous with the procedure in order to guarantee that a 22-bit pc-relative displacement in the check instruction will reach the target. If the recovery code is distant, the target of the check instruction may be a small piece of "trampoline" code that branches to the recovery code.

If a speculative load is issued to an unaligned address, the OS may deliver a NaT. An application cannot expect to use a user-level trap handler to emulate the unaligned load unless the code is compiled with recovery code.

9.3 Multi-threaded code

In multi-threaded applications, the use of the volatile type qualifier should be interpreted to mean that the variables designated with that type may be modified asynchronously by any thread. The compiler must observe ordering

restrictions with respect to loads and stores, and should not remove otherwise unnecessary memory references to these variables.

In addition, the compiler must generate appropriate ordered load and store instructions to prevent the hardware from executing volatile references out of order. All loads to a volatile type must use acquire semantics (using the ".acq" completer), and all stores to a volatile type must use release semantics (using the ".rel" completer). These completers ensure that no load will complete prior to an earlier load with acquire, and no store will complete prior to a subsequent store with release.

The use of a memory fence operation prior to a load with acquire implements stronger ordering, but is not required by these conventions.

9.4 Setjmp and longjmp

The contents of a procedure's register stack frame are not preserved in a jump buffer by a call to setjmp. If the compiler has a temporary value live in a stacked register before the call to setjmp, with a subsequent use after the call to setjmp, that value will not be saved and restored by a setjmp/longjmp. Instead, after a longjmp, the register will have whatever value it had at the point in time when longjmp was called. If the original value reaches all subsequent call points in the procedure, the code will behave as expected. If the register is reused or otherwise modified, however, the value in that register following a longjmp is unpredictable.

To keep a stacked register live across a call to setjmp, the compiler can do one of three things: (1) dedicate that register for the rest of the procedure, (2) copy it to a real preserved register (r4–r7), or (3) spill it to a dedicated memory stack location. Alternatively, the compiler can simply rematerialize it after the call to setjmp.

9.5 Up-level referencing

Local variables visible to nested procedures must be saved in memory at any procedure call or exception control point; a procedure's local registers are not visible to its nested procedures.

These conventions suggest, but do not require, the use of a static link passed as an implicit parameter to nested procedures. The static link can be used by the nested procedure to access local variables in its enclosing scope. The rules for forming and passing static links are as follows:

- A level-one procedure (outermost) calling a level-two procedure should pass, as the static link, the address of a known reference point within its stack frame (for example, its frame pointer).
- A nested procedure calling another procedure at the same level should pass, as the static link, the static link that it received.

- A nested procedure calling a procedure nested within it should store the static link that it received at a known place within its own stack frame, then pass, as the static link to the new procedure, the address of a known reference point within its own stack frame (for example, a pointer to the static link that it saved).
- A nested procedure calling a less-deeply nested procedure must follow the chain of static links to obtain the correct static link to pass.

When forming function pointers that refer to nested procedures, the same rules apply. The static link must be determined at the time the function pointer is materialized, and stored with the function pointer.

To reference local variables in enclosing scopes, the chain of static links must be followed to obtain a pointer to the enclosing scope's stack frame. The compiler can determine statically the offset of the desired local variable relative to the reference point used for the static link.

An alternate implementation is a display pointer, also passed as an implicit parameter to each nested procedure.

9.6 C++ conventions

The "this" pointer is passed as an implicit first parameter to all non-static class member functions.

Any object that requires a copy constructor must be passed by copy-reference rather than by value (that is, the compiler must copy it to a temporary location in memory and pass the address of this location in the argument list). This guarantees that the object will have a valid memory address as required by the copy constructor. The temporary location should be in the caller's memory stack frame.

Chapter 10

Context Management

10.1 Process/thread context

The following table lists the resources that constitute the context that is visible to the user-mode process or thread (not including the program's address space). These are the registers that must be saved and restored on an asynchronous context switch (i.e., a context switch triggered by an outside event, such as a signal). For a synchronous context switch (i.e., a direct call to a context-switch routine), scratch registers do not need to be saved.

Table 10–1. Resources to be saved on context switches

Resource	*Synchronous*	*Asynchronous*
Instruction pointer (ip)	•	•
Global data pointer (gp)	•	•
Stack pointer (sp)	•	•
Thread pointer (tp)	•	•
Backing store pointer (ar.bsp/ar.bspstore)	•	•
Floating-point status register (ar.fpsr)	•	•
RSE NaT collection register (ar.rnat)	•	•
User NaT collection register (ar.unat)	•	•
Previous function state (ar.pfs)	•	•
Current frame marker		•
RSE control register (ar.rsc)		•
Loop counter (ar.lc)	•	•
Epilogue counter (ar.ec)		•
Compare and exchange comparison value (ar.ccv)		•
Preserved general registers (r4–r7) (including NaT bits)	•	•
Scratch general registers (r2–r3, r8–r11, r14–r31) (including NaT bits)		•
Preserved floating-point registers (f2–f5, f16–f31)	•	•

Table 10–1. Resources to be saved on context switches

Resource	*Synchronous*	*Asynchronous*
Scratch floating-point registers (f6–f15, f32–f127)		•
Preserved predicate registers (p1–p5, p16–p63)	•	•
Scratch predicate registers (p6–p15)		•
Preserved branch registers (b1–b5)	•	•
Scratch branch registers (b0, b6–b7)		•

Note *The User NaT collection register must be saved separately from the NaT bits for the general registers, since it contains the NaT bits for preserved general registers that a procedure has spilled on behalf of its caller. This register must be saved before any general registers are saved, as the saving of general registers writes to this register. Once the general registers have been saved as part of the state save procedure, the User NaT collection register will contain the NaT bits for the newly-saved registers, and can then be saved again.*

10.2 User-level thread switch, coroutines

Thread switches and coroutine calls can be done with a procedure call, so no scratch registers need to be saved as part of the context. The first part of this routine saves the current thread context on the stack:

1. Save ar.rsc, ar.bsp and ar.pfs.
2. Use flushrs instruction to flush dirty registers to the backing store.
3. Set the RSE in enforced lazy mode by clearing both rsc.mode bits.
4. Save ar.rnat and other registers that must be saved for a synchronous context switch.

At this point, the RSE is frozen, and all dynamic registers up to the current procedure frame are saved in the backing store. We can now change the memory stack pointer (sp) to point to the new thread's stack, and restore the new thread's context from there:

1. Invalidate the ALAT using the invala instruction.
2. Restore ar.bspstore (the saved ar.bsp).
3. Restore ar.rnat and ar.pfs.
4. Restore ar.rsc. If eager loads are enabled, it will begin restoring dynamic registers from previous stack frames. Otherwise, it will restore registers from the backing store when needed for a return branch.
5. Restore the remaining preserved registers.
6. Return to the new thread.

10.3 Setjmp/longjmp

The jmpbuf structure declared in <jmpbuf.h> needs to contain the state listed in Table 10–1 for a synchronous context switch. The instruction pointer (ip) is the return BR from the call to setjmp. It must also contain the signal mask (for sigsetjmp/siglongjmp).

Saving and restoring this context is similar to a thread switch. A longjmp must also invalidate the ALAT.

When the NaT bits for the preserved registers are saved, care should be taken that the representation is not dependent on the address of the jump buffer itself: the st8.spill instruction saves the NaT bit in the user NaT collection register based on the memory address.

Note *The user NaT collection register is itself a preserved register, and must be saved in the jump buffer before any preserved general registers are spilled. The saved copy of the user NaT collection register should not be adjusted for the jump buffer address—this adjustment should be made only for the NaT bits resulting from the stores of the preserved registers into the jump buffer.*

Chapter 11

Stack Unwinding and Exception Handling

Stack unwinding is the process of tracing backwards through a process' stack of activation records. Every procedure in an EM program has at least a frame on the register stack, and may also have a frame on the memory stack. In order to print a stack trace, debuggers require the ability to identify every frame on these stacks, and to show the process context associated with each one. Exception handling often requires the ability to remove a number of frames from the stack and to transfer control to an exception handling routine that may have been far down the stack.

For the register stack, the ar.pfs register contains sufficient information to identify the previous frame, given the state of the current register stack frame. This works for only one level of nesting, however, since there is no architected stack of ar.pfs registers. Thus, in order to unwind the register stack, we must impose a convention for saving and recovering the ar.pfs register in each frame.

For the memory stack, there is no architected mechanism for recording the sp value for each stack frame, or for associating memory stack frames with register stack frames. While different procedures will need differently-sized stack frames, we expect that most procedures will allocate a frame whose size does not change while the procedure is active. Thus, for most procedures, we can simply record this fixed frame size in a static table, and use the instruction pointer (IP) as a key to this table. For procedures whose frames can vary in size, we must impose a convention for saving and recovering the sp value for the previous frame on the stack.

As the stacks are unwound, it is also necessary to recover the values of preserved registers that were saved by each procedure in the activation stack, so that debuggers have access to correct values of local variables, and so that exception handlers can operate correctly. This requirement also imposes conventions for saving and recovering the values of these preserved registers.

In all cases, we wish to retain as much flexibility as possible for the compiler in its use of registers and code generation. Thus, these conventions allow the compiler to save the necessary values in a variety of locations, and with a variety of code sequences. We use the IP as a key for locating an unwind table entry that describes everything necessary for locating the previous register and memory stack frames, as well as the previous IP. The compiler is responsible for generating this static unwind table entry for each procedure that it generates code for.

In most operating environments, unwinding the stack will be done via an unwind library that can be called from the process itself, from a debugger, or for exception handling. It operates on context records; the primary routine reconstructs the context for a previous frame given the context for its descendent frame. Because the structure of a context record, and the interface between the OS and exception handling mechanism is environment dependent, this unwind library is also environment-dependent, and is not defined as part of the runtime architecture. This chapter describes the framework for unwinding the stack and for processing exceptions, including the format of the static unwind tables constructed by the compilers, and the code generation conventions imposed as a result.

11.1 Unwinding the stack

The process of unwinding the stack begins with an initial context record describing the process state in the most recent procedure activation, at the point of interruption. From this initial state, the stack is unwound one procedure frame at a time, using static information generated by the compilers about each procedure to help it reconstruct a context record describing the previous procedure, which is suspended at a point just after the procedure call or an asynchronous interruption.

Initial context

Every stack unwind starts with an initial context, obtained from one of three sources:

- The debugger. The context record is obtained from the OS through the debugging API.
- The unwind library. The context is constructed as for the first half of a user-mode thread switch.
- From exception handler. The context is constructed by the OS and passed to the exception handler.

Step to previous frame

This process builds a context record corresponding to the next older frame on the stack. This context record can, in turn, be used to unwind to the next frame. The following steps will reconstruct the context for the previous frame:

1. Find the return link in the current context, and set IP in the previous context to that address.
2. Find the previous frame marker in the current context (e.g., in the ar.pfs register), and copy it to the current frame marker (cfm) in the previous context.
3. Determine the value of gp for the new IP, and set gp in the previous context to that value.
4. Set sp in previous context to sp from current context plus the current memory frame size.

5. Set ar.bsp in the previous context to ar.bsp from the current context minus size of the input/local region of the frame (taking NaT collections that may have been saved to the backing store into account). The frame size can be calculated from the frame marker.
6. Find the saved copies of the preserved registers in the current context, and copy them to the previous context.

The bottom of the call stack is identified by a saved return link of 0.

The information needed to execute these steps correctly is recorded by the compilers in static unwind information, stored in the text segment of the program itself. The structure of this information is described in Section 11.4. Each text segment contains a table of unwind information, and the dynamic loader is expected to provide an API for finding the unwind table, given a known IP. This API is specific to the operating environment, and is not described here.

When a process is delivered an asynchronous interruption (via a mechanism that is environment dependent), the full process context needs to be saved so that the process can continue executing correctly once the interruption has been handled. Typically, this context will be saved on the memory stack, and a new procedure frame will be constructed for the interruption handler. The first procedure frame in the interruption processing must be marked in such a way that the unwind routine can recognize that unwinding past the point of interruption requires a restoration of the full context. This, unfortunately, is also an environment-dependent operation, and cannot be described in the runtime architecture.

When the operating system delivers a context to the application, it may be necessary for the register stack backing store to be split into two or more non-contiguous pieces. An application that examines its backing store should be prepared to deal with this; this also is an environment-dependent operation.

11.2 Exception handling framework

The exception handling model for EM is partitioned into a language-independent component and a language-dependent component. The language-independent component is responsible for fielding an exception, searching for an exception handler, and unwinding the stack prior to processing an exception. Each source language that supports exception handling must provide, as part of its runtime library, a "personality" routine that implements the language-dependent component of this model.

This document uses the C++ exception handling mechanism as an example of the language-dependent component. The description of the C++-specific data structures and routines should be treated as an example, rather than a specification of the C++ design. Text that discusses language-specific implementation appears indented and italicized like this paragraph.

The exception handling model is oriented around procedure frames on the memory and register stacks. Each frame corresponds to an activation of a procedure, which may or may not have associated exception handling

requirements. A procedure may have two kinds of exception handling requirements:

- It may allocate some objects that require deallocation or some other form of cleanup if the procedure or any of its blocks are terminated abnormally.
- It may have one or more *try regions*, which are regions of code that specify an action to be taken if an exception occurs while control is within them.

In either of these cases, the compiler records the requirements in the static unwind information for the procedure, and stores a reference to the personality routine for that procedure. Typically, a language will use a single personality routine for all procedures, but this is not a requirement (for example, a language may define a separate personality routine for procedures that require cleanup, but have no try regions.)

Try regions may be nested both statically, within the procedure, and dynamically, through procedure calls. When an exception occurs, each try region is inspected to determine if it has specified an action for that particular exception. The try regions are inspected in order, beginning with the innermost region.

> *In C++, a* try/catch *statement defines a try region, and the filter controls which exceptions are to be caught and handled within that region.*

Exceptions are raised by invoking a routine in the language-independent component called the *exception dispatcher*, which initiates the process of handling the exception. Synchronous exceptions may be raised directly by the application through a language-specific construct; asynchronous exceptions may be raised in response to hardware-detected traps or faults.

> *In C++, synchronous exceptions can be raised with the* throw *statement. This statement creates an exception object, which is matched against the prototype in each* catch *clause for each active* try *statement. C++ does not define asynchronous exceptions.*

The dispatcher unwinds each frame on the stack non-destructively, beginning with the topmost frame, searching for frames with one or more try regions. For each frame that has exception handling information, the dispatcher invokes the personality routine, which determines which try regions, if any, are currently active. For each active try region, starting with the most deeply nested one, the personality routine determines whether to dismiss the exception, handle it, or continue the search with the next try region, or with the previous frame on the stack. If the personality routine does find a try region with a handler for the exception, it invokes the unwinder to unwind the stack a second time. During this second unwind, the unwinder invokes the personality routines for each frame again so that cleanup actions may be executed as necessary. When the unwind reaches the frame that contains the exception handler, control is tranferred to the handler.

The relationships among these components are illustrated in Figure 11–1. The shaded boxes identify the components that are specific to C++.

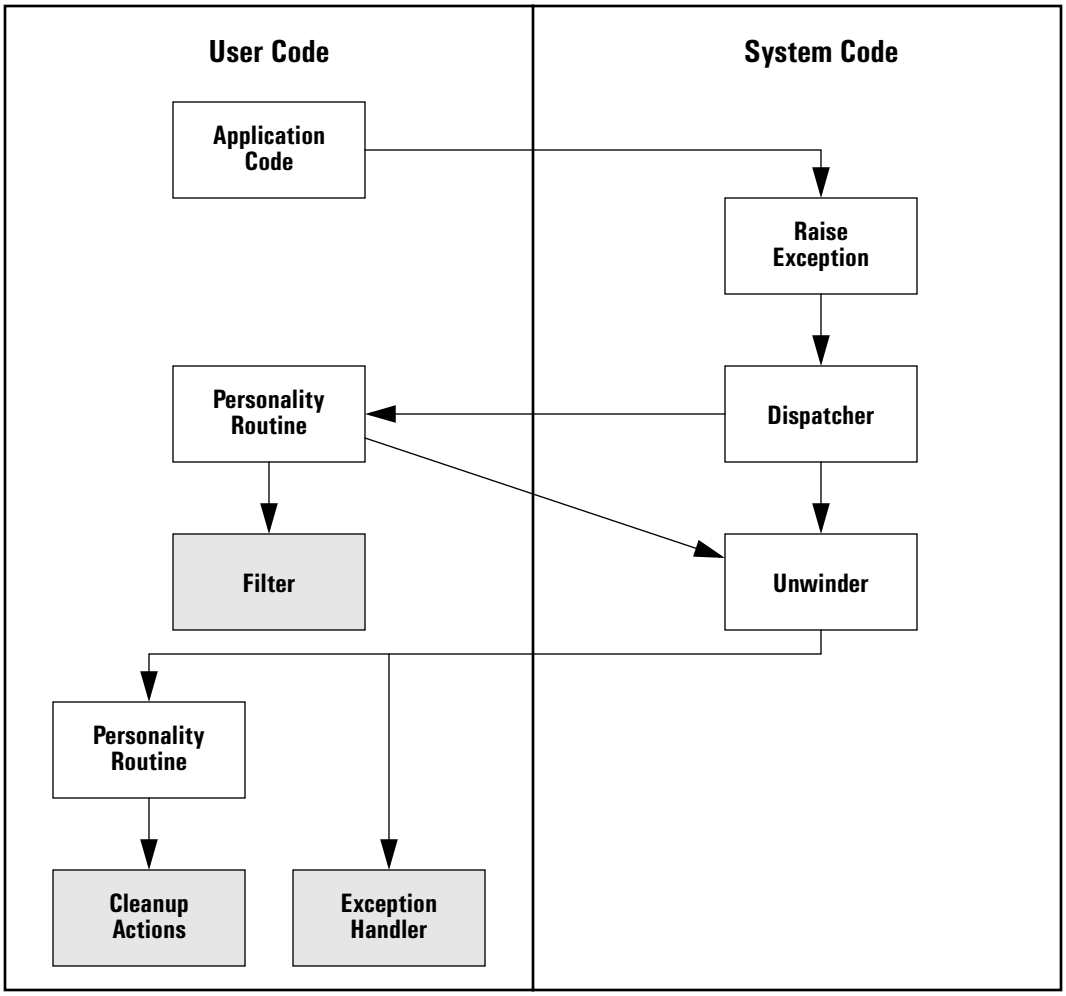


Figure 11–1. Components of the exception handling mechanism

11.3 Coding conventions for reliable unwinding

This section describes the coding conventions that must be observed to guarantee unwindability from every point in the program. For the purposes of unwinding, we divide every procedure up into one or more regions, which are classified as either “prologue” or “body” regions.

A “prologue” region is one where the register stack and memory stack frames are established and where key registers are saved. In order to unwind correctly when the IP is is one of these regions, the unwinder must have a detailed description of the order of operations within the region, so that it knows what state has changed, and which registers have been saved at any given point in that region.

A “body” region does not change the state of either stack frame, and does not save any additional preserved registers. Thus, the unwinder needs to know only the state of the frame for the entire region, and the relative location of the IP within the region is irrelevant.

For both types of regions, the unwinder needs to know the state of the stack frames and preserved registers upon entry to the region. There are four ways to establish the entry state for an unwind region:

- The first region in the procedure assumes that both stack frames are unallocated, and no registers have been saved upon entry to the region.
- A region may modify the state of the stack frames and preserved registers; each subsequent region takes the previous region's exit state as its entry state.
- When control does not flow into a region from directly above it, the region may specify an alternate predecessor region whose exit state is used instead.
- Zero-length prologue regions may be inserted just prior to a prologue or body region to set up the correct entry state.

Regions may begin and end at arbitrary instructions, without regard to bundle boundaries or cycle breaks.

Conventions for prologue regions

A typical prologue region will do some or all of the following steps:

- Allocate a new register stack frame. The placement of this step is not important to the unwind process (although it must precede any other operations in the prologue that require the use of local stack registers).
- Allocate a new memory stack frame. For fixed-size frames, the stack pointer (sp) must be modified in a single instruction (either with a single add immediate, or by performing intermediate calculations in a scratch register before modifying sp). The location of this instruction and the fixed frame size must be recorded in the unwind descriptor. For variable-size frames, the stack pointer must be saved in a general register that is kept valid throughout the remainder of the prologue region and the following body region(s). This copy of the previous stack pointer is called psp. The location of the copy instruction, and the GR number must be recorded in the unwind descriptor.
- Save the previous function state (ar.pfs), either in a general register or on the memory stack. The location of this instruction, and the GR number or stack offset must be recorded in the unwind descriptor. Normally, the previous function state is copied to a GR by the alloc instruction that allocates a new register stack frame. If the previous function state is to be stored in the memory stack, however, the location of the instruction that stores the GR to memory should be recorded, and the original pfs may not be destroyed until after the store.
- Save the return pointer (rp), either in a general register or on the memory stack. The location of this instruction, and the GR number or stack offset must be recorded in the unwind descriptor. Saving to the memory stack requires two steps—one to copy it to a GR, and another to store it; the location of the store is the one to record, and the original rp may not be destroyed before the store.
- Save preserved registers, either on the memory stack or in local registers in the current register stack frame. In general, the location of each instruction used to save a preserved register, and the GR number or stack

offset must be recorded. There are five groups of preserved registers: GRs, FRs, BRs, predicates, and ARs (ar.unat , ar.rnat, ar.lc, ar.fpsr, ar.bsp, and ar.bspstore). The predicates must be copied as a whole to a GR with a single Move from Predicates instruction; if they are to be stored on the memory stack, the Store instruction is the one to record. Any arbitrary subset of preserved GRs, FRs, and BRs may be saved in a prologue, but they must be saved in ascending order by register number within each group (saves from different groups may be interleaved). Saving a BR to memory (other than rp) requires two steps—a move to GR, and a store; the location of the store is the one to record, and the value of the BR may not be destroyed until the store is completed.

The unwinder must also know where preserved registers are saved in the memory stack frame, because it needs to reconstruct the values of these registers as it unwinds the stack. The conventions for the spill area are discussed below.

A prologue region may also contain any amount of other code that is irrelevant to the unwind process. For better efficiency during the unwind process, however, the size of the prologue region should be kept as small as possible, and it should be defined to end immediately after the last of the above steps.

Prologue regions may occur in the interior of a procedure. These typically represent register spill sequences that have been "shrink-wrapped" into a small block of conditional code.

The encoding of the unwind descriptors for prologue regions recognizes several common cases that reduce the size of the unwind information significantly. Compilers are encouraged to observe these conventions for low optimization levels and whenever it would not adversely affect the quality of optimization. These cases include:

- The prologue saves rp, ar.pfs, psp, and the predicates (as needed) in consecutive registers in the ins/locals area of the current register stack frame.
- The prologue saves all of its subset of preserved registers before modifying any of them. In this case, the locations of individual save instructions do not need to be recorded, and the restrictions on their relative ordering are eliminated.
- A leaf procedure that does not create a memory stack frame or save any preserved registers does not require any unwind descriptors.

Conventions for body regions

In general, body regions may do anything that does not invalidate the state of the stack frames and preserved registers as recorded for that region. In particular, a body region must obey the following restrictions:

- If the memory stack frame is fixed size, it may not modify the sp register.
- If the memory stack frame is variable size, it may modify sp at any point, but it may not destroy the psp value, which must be undisturbed in a known place throughout the region.

- It may not destroy the previous frame marker in its known location. If the previous frame marker is still in ar.pfs, this means that there may not be any calls in the region.
- It may not destroy the return IP in its known location. If the return IP is still in rp, this means that there may not be any calls in the region.
- It may not destroy any preserved registers that have not been saved prior to entry to the region.
- It may not destroy the saved copies of any preserved registers that have been saved prior to entry to the region.

A body region may restore ar.pfs, rp, and any preserved registers, as long as the saved copies remain valid. Thus, the unwinder does not need a specific "epilogue" region that is distinct from the body region.

The memory stack pointer (sp) is typically restored just before executing a return branch. In a normal epilogue at the end of a body region, the compiler may place the instruction that restores the previous sp value anywhere within a few instructions of the end of the region; the unwind descriptor format provides a place to record the exact location of this instruction. If the procedure has a memory stack frame, and has returns in the middle of the body, the compiler must separate the procedure into separate body regions, each ending at the point of each return.

Conventions for the spill area in the memory stack frame

The spill area for preserved general registers, floating-point registers, and branch registers is near the base of the stack frame, in a continuous range ending, by default, at the base of the stack frame plus 16 bytes (psp+16). In other words, the 16-byte scratch area in the caller's stack frame normally contains the last 16 bytes of the spill area. If the scratch area is needed for saving register parameters for a variable-argument list procedure, the spill area may be moved so that it ends at a lower address, but the ending address must be a fixed location relative to the base of the frame (psp).

Locations in the spill area are reserved for each preserved GR, FR, and BR that is saved anywhere within the procedure (including shrink-wrapped regions). Locations are allocated, from low address to high, first for general registers, then for branch registers, and finally for floating-point registers. Registers are saved in numerical order, lower-numbered registers at lower addresses. The spill area must end at a 16-byte boundary, so that all the floating-point spill locations are 16-byte aligned.

It is not required that all registers preserved in the spill area be consecutive from each register file. If, for example, GR 4 and GR 7 are preserved, but GR 5 and GR 6 are not, space is allocated only for GR 4 and GR 7.

A compiler may need to spill scratch registers in addition to preserved registers. There are no required conventions for spilling scratch registers, since they do not need to be recovered during a stack unwind. It is expected, however, that general register spills will be adjacent to the preserved general register spill area in order to make the best use of the User NaT collection register.

Normally, the unwinder expects to find the NaT bits for the preserved registers in the User NaT collection register, ar.unat. If the total spill area for general registers (scratch and preserved combined) exceeds 64 double-words, the compiler may be forced to save the User NaT collection register in order to spill up to an additional 64 general registers. In this overflow situation, the compiler must manage two or more NaT collections by swapping them in and out of the single collection register. The NaT collection that contains the NaT bits for the preserved registers is called the "primary unat collection," and the unwinder must know where to find these bits. In procedures where the NaT collection register is multiplexed, the compiler must record the location of the primary unat collection in the unwind information.

11.4 Data structures

The exception handling mechanism uses three data structures:

- An unwind table, which allows the dispatcher and unwinder to associate an IP value with a procedure and its unwind and exception handling information. Every procedure that has either a memory stack frame or exception handling requirements, or both, has one entry in this table. (If the compiler has generated more than one non-contiguous region of code for a procedure, there will be one entry in this table for each region.) Each unwind table entry points to an information block that contains the other two data structures.
- A set of unwind descriptors for each procedure.
- An optional language-specific data area for each procedure.

The dispatcher and unwinder both use the unwind table to locate an unwind entry for a procedure, given an IP value. The unwinder also uses the unwind descriptor list so that it can properly unwind the stack from any point in the procedure.

The language-specific data area is used to store cleanup actions and a try region table.

11.4.1 Unwind table

The unwind table entries contain three fields, as illustrated in Figure 11–2; each field is a 64-bit doubleword. The first two fields define the starting and ending addresses of the procedure, respectively, and the third field points to a variable-size information block containing the unwind descriptor list and language-specific data area. The ending address is the address of the first bundle beyond the end of the procedure. These values are all segment-relative offsets, not absolute addresses, so they do not require run-time relocations. The unwind table is sorted by the procedure start address. The shaded area in the figure represents the language-specific data area.

If a leaf procedure has no stack frame, has no exception handling requirements, and keeps its return pointer in b0, no unwind table entry is necessary for the procedure. The unwinder must assume these conditions when the IP does not correspond to any procedure table entry.

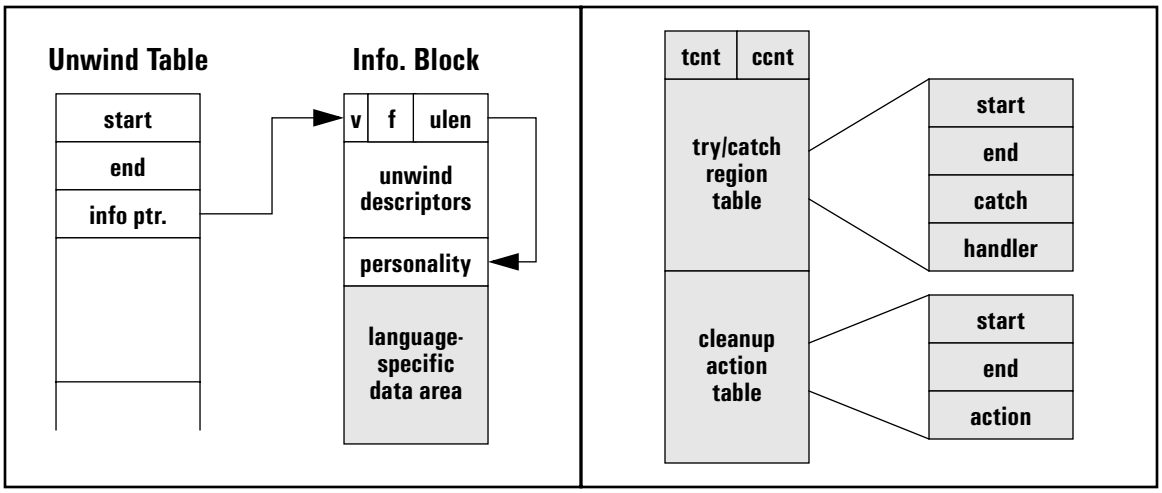


Figure 11–2. Unwind table and example of language-specific data area

The first doubleword of the information block consists of three fields: a 16-bit version number for the unwind descriptors, 16 flag bits, and a 32-bit length field. These fields may be accessed with the following macros:

```
#define UNW_VER(x)              ((x) >> 48)
#define UNW_FLAG_MASK           0x0000ffff00000000L
#define UNW_FLAG_OSMASK         0x0000f000000000000L
#define UNW_FLAG_EHANDLER(x)    ((x) & 0x0000000100000000L)
#define UNW_FLAG_UHANDLER(x)    ((x) & 0x0000000200000000L)
#define UNW_LENGTH(x)           ((x) & 0x00000000ffffffffL)
```

The unwind version number identifies the version of the unwind descriptor format. For this specification, the version number is 1.

The unwind length field identifies the length (in doublewords) of the unwind descriptor area.

Two flag bits are currently defined, and the four defined by UNW_FLAG_OSMASK are reserved for implementation-specific use; the remaining bits are reserved for future use. The EHANDLER flag is set if the personality routine should be called during search for an exception handler. The UHANDLER flag is set if this routine should be called during the second unwind. If neither bit is set, there is no frame handler for this procedure, and the personality routine identifier should be omitted, along with the entire language-specific data area.

> *In C++, the* EHANDLER *bit is set if the procedure contains any* try/catch *regions, and the* UHANDLER *bit is set if there are any cleanup actions.*

The personality routine identifier is accessed by adding the size of the unwind descriptor area (*ulen*, which is count of doublewords, not bytes), plus the size of the header doubleword, to the information block pointer. This identifier contains the 64-bit gp-relative offset of a doubleword in the linkage table that contains a function pointer, which in turn points to the function descriptor of the personality routine. The function pointer itself must be in the data segment because it may need relocation. The dispatcher should call this routine during the first unwind only if the EHANDLER bit is set, and during the second unwind only if the UHANDLER bit is set. The language-specific data

immediately follows the personality routine identifier, so the address of this area must be made available to the personality routine.

11.4.2 Unwind descriptor area

The unwind descriptor area contains a contiguous sequence of records describing the unwind regions in the procedure. Each group of records begins with a region header record identifying the type and length of the region. The region header record is followed by any number of descriptor records that supply additional unwind information about the region.

The unwind descriptor records are divided into three categories: region header records, descriptor records for prologue regions, and descriptor records for body regions. This section describes the record types in each of these categories, lists rules for using unwind descriptor records, and explains how the records should be processed.

The information is encoded in variable-length records with a record type and one or more additional fields. The length of each record is implicit from the record type and its fields. All records are an integral number of bytes in length. In the descriptor record tables in the next three sections, the third column lists the format of each record type. These record formats are described in Appendix B.

Since the unwind descriptor area must be a multiple of 8 bytes, the last unwind descriptor must be followed by zero bytes as necessary to pad the area to an 8-byte boundary. These zero bytes will be interpreted as prologue region header records, specifying a zero-length prologue region, and serve as no-ops.

Region header records

The region header records are listed in Table 11–1.

Table 11–1. Region Header Records

Record Type	*Fields*	*Format*	*Description*
prologue	rlen	R1/R3	Defines a general prologue region.
prologue_gr	rlen, mask, grsave	R2	Defines a prologue region with a mask of saved registers, and a set of GRs used for saving preserved registers.

The fields in these records are used as follows:

- **rlen** contains the length of the region, measured in instruction slots (three slots per bundle, counting X-unit instructions as two slots).
- **mask** indicates which registers are saved in the prologue. The prologue_gr region type is used for entry prologues that save one or more preserved registers in the local register area of the register stack frame. This field defines what combination of rp, ar.pfs, psp, and the predicates are preserved in standard GRs in the local area of the register stack frame. This mask is four bits; see Appendix B for the allocation of these bits. Other registers may be preserved in the prologue, but additional descriptor records are required for registers other than these four.

- **grsav** identifies the first GR used to save the preserved registers identified in the mask field. Normally, this should identify a register in the procedure's local stack frame (i.e., it should be greater than or equal to 32). Leaf procedures, however, may choose to use any consecutive sequence of scratch registers.

By default, the entry state for a region is assumed to match the exit state of the preceding region. The exit state of a region is determined as follows:

- For prologue regions, the exit state is the logical combination of the entry state and the state modifications performed by the prologue. The descriptor records following the region header record describe these modifications.
- For body regions with no epilogue code, the exit state is the same as the entry state.
- For body regions with epilogue code, the exit state is the same as the entry state of the corresponding prologue whose effect is being undone. When shrink-wrap regions are nested, it is possible to reverse the effects of multiple prologues at once.

Descriptor records for prologue regions

This section lists the descriptor records that may be used to describe prologue regions. In the absence of any descriptor records or information in the region header record, a prologue is assumed to create no memory stack frame and save no registers. Descriptors need to be supplied only to override these defaults.

The following descriptor records are used to record information about the stack frame and the state of the previous stack pointer (psp).

Table 11–2. Prologue Descriptor Records for the Stack Frame

Record Type	*Fields*	*Format*	*Description*
mem_stack_f	t, size	P7	Specifies a fixed-size memory stack frame, when sp is modified, and size of frame.
mem_stack_v	t	P7	Specifies a variable-size memory stack frame, and when psp is saved.
psp_gr	gr	P3	Specifies GR where psp is saved.
psp_sprel	spoff	P7	Specifies memory location where psp is saved, as an sp-relative offset.

The fields in these records are used as follows:

- **t** describes a time, *t*, when a particular action occurs within the prologue. The time is specified as an instruction slot number, counting three slots per bundle. The first instruction slot in the prologue is numbered 0. For procedures with a memory stack frame, the instruction that modifies sp (fixed-size frame) or that saves psp (variable-size frame) must be identified with either a mem_stack_f or a mem_stack_v record. In all

other cases, if the time is not specified, the unwinder may assume that the original contents of the register is valid through the end of the prologue, and that the saved copy is valid by the end of the prologue.

- **size** contains the fixed size of the memory stack frame, measured in 16-byte units.
- **gr** identifies a general register, or the first in a consecutive group of general registers, that is used for preserving the value of another register (as implied by the record type). Typically, this field will identify a general register in the procedure's local stack frame. A leaf procedure, however, may choose to use scratch registers. (A non-leaf procedure may also use scratch registers through a body region that makes no calls, but it would need to move any values saved in scratch registers to a more permanent save location prior to making any calls. It would need a second prologue region to describe this movement.)
- **spoff** identifies a location in the memory stack where a register or group of registers are spilled to memory. This location is specified relative to the current stack pointer. See Appendix B for the encoding of this field.

The following descriptor records are used to record the state of the return pointer (rp).

Table 11–3. Prologue Descriptor Records for the Return Pointer

Record Type	*Fields*	*Format*	*Description*
rp_when	t	P7	Specifies when rp is saved.
rp_gr	gr	P3	Specifies GR where rp is saved.
rp_br	br	P3	Specifies alternate BR used as return pointer.
rp_psprel	pspoff	P7	Specifies memory location where rp is saved, as a psp-relative offset.
rp_sprel	spoff	P8	Specifies memory location where rp is saved, as an sp-relative offset.

The fields in these records are used as follows:

- **br** identifies a branch register that contains the return link, when the return link is not either in b0 or saved to another location.
- **pspoff** identifies a location in the memory stack where a register or group of registers are spilled to memory. The location is specified relative to the previous stack pointer (which is equal to the current stack pointer plus the frame size). See Appendix B for the encoding of this field.

The following descriptor records are used to record the state of the previous function state register (ar.pfs).

Table 11–4. Prologue Descriptor Records for the Previous Function State

Record Type	*Fields*	*Format*	*Description*
pfs_when	t	P7	Specifies when ar.pfs is saved.
pfs_gr	gr	P3	Specifies GR where ar.pfs is saved.

Table 11–4. Prologue Descriptor Records for the Previous Function State

Record Type	*Fields*	*Format*	*Description*
pfs_psprel	pspoff	P7	Specifies memory location where ar.pfs is saved, as a psp-relative offset.
pfs_sprel	spoff	P8	Specifies memory location where ar.pfs is saved, as an sp-relative offset.

The following descriptor records are used to record the state of the preserved predicates.

Table 11–5. Prologue Descriptor Records for Predicate Registers

Record Type	*Fields*	*Format*	*Description*
preds_when	t	P7	Specifies when the predicates are saved.
preds_gr	gr	P3	Specifies GR where predicates are saved.
preds_psprel	pspoff	P7	Specifies memory location where predicates are saved, as a psp-relative offset.
preds_sprel	spoff	P8	Specifies memory location where predicates are saved, as an sp-relative offset.

The following descriptor records are used to record the state of the preserved general registers, floating-point registers, and branch registers.

Table 11–6. Prologue Descriptor Records for GRs, FRs, and BRs

Record Type	*Fields*	*Format*	*Description*
fr_mem	rmask	P6	Specifies which preserved floating-point registers are spilled to memory by this prologue, as a bit mask.
frgr_mem	grmask, frmask	P5	Specifies which preserved general and floating-point registers are spilled to memory by this prologue, as a bit mask.
gr_gr	grmask, gr	P9	Specifies which preserved general registers are saved in other general registers, as a bit mask, and GR where first preserved GR is saved.
gr_mem	rmask	P6	Specifies which preserved general registers are spilled to memory by this prologue, as a bit mask.
br_mem	brmask	P1	Specifies which preserved branch registers are spilled to memory by this prologue, as a bit mask.
br_gr	brmask, gr	P2	Specifies which preserved branch registers are saved in general registers by this prologue, as a bit mask, and GR where first BR is saved.

Table 11–6. Prologue Descriptor Records for GRs, FRs, and BRs

Record Type	*Fields*	*Format*	*Description*
spill_base	pspoff	P7	Specifies base of spill area in memory stack frame, as a psp-relative offset.
spill_mask	imask	P4	Specifies when preserved registers are spilled, as a bit mask.

The fields in these records are used as follows:

- **rmask, frmask, grmask, brmask** identify which preserved FRs, GRs, and BRs are saved by the prologue region. The fr_mem record uses a short rmask field, which can be used when a subset of floating-point registers from the range f2–f5 is saved. The frgr_mem record can be used for any number of saved floating-point and general registers. The gr_mem record can be used when only general registers (r4–r7) are saved.
- **imask** identifies when each preserved FR, GR, and BR is saved. It contains a two-bit field for each instruction slot in the prologue, indicating whether the instruction in that slot saves one of these preserved registers. The length of this field is implied by the size of the prologue region as given in the region header record. It contains two bits for each instruction slot in the region, and the length of the field is rounded up to the next whole byte boundary.

If a prologue saves one or more preserved FRs, GRs, or BRs, and the spill_mask record is omitted, the unwinder may assume that the original contents of those preserved registers are valid through the end of the prologue, and that the saved copies are valid by the end of the prologue.

There may be only one spill_base and one spill_mask record per prologue region.

Each gr_gr and br_gr record describes a set of registers that is saved to a consecutive set of general registers (typically in the local register stack frame). To represent registers saved to non-consecutive general registers, two or more of each of these records may be used.

The following descriptor records are used to record the state of the User NaT Collection register (ar.unat).

Table 11–7. Prologue Descriptor Records for the User NaT Collection Register

Record Type	*Fields*	*Format*	*Description*
unat_when	t	P7	Specifies when ar.unat is saved.
unat_gr	gr	P3	Specifies GR where ar.unat is saved.
unat_psprel	pspoff	P7	Specifies memory location where ar.unat is saved, as a psp-relative offset.
unat_sprel	spoff	P8	Specifies memory location where ar.unat is saved, as an sp-relative offset.

The following descriptor records are used to record the state of the Loop Counter register (ar.lc).

Table 11–8. Prologue Descriptor Records for the Loop Counter Register

Record Type	*Fields*	*Format*	*Description*
lc_when	t	P7	Specifies when ar.lc is saved.
lc_gr	gr	P3	Specifies GR where ar.lc is saved.
lc_psprel	pspoff	P7	Specifies memory location where ar.lc is saved, as a psp-relative offset.
lc_sprel	spoff	P8	Specifies memory location where ar.lc is saved, as an sp-relative offset.

The following descriptor records are used to record the state of the floating-point status register (ar.fpsr).

Table 11–9. Prologue Descriptor Records for the Floating-Point Status Register

Record Type	*Fields*	*Format*	*Description*
fpsr_when	t	P7	Specifies when the floating-point status register is saved.
fpsr_gr	gr	P3	Specifies GR where the floating-point status register is saved.
fpsr_psprel	pspoff	P7	Specifies memory location where the floating-point status register is saved, as a psp-relative offset.
fpsr_sprel	spoff	P8	Specifies memory location where the floating-point status register is saved, as an sp-relative offset.

The following descriptor records are used to record the state of the primary unat collection.

Table 11–10. Prologue Descriptor Records for the Primary unat Collection

Record Type	*Fields*	*Format*	*Description*
priunat_when_gr	t	P8	Specifies when the primary unat collection is copied to a GR.
priunat_when_mem	t	P8	Specifies when the primary unat collection is saved in memory.
priunat_gr	gr	P3	Specifies GR where the primary unat collection is copied.
priunat_psprel	pspoff	P8	Specifies memory location where the primary unat collection is saved, as a psp-relative offset.
priunat_sprel	spoff	P8	Specifies memory location where the primary unat collection is saved, as an sp-relative offset.

The following descriptor records are used to record the state of the backing store, when it is necessary to record a discontinuity.

Table 11–11. Prologue Descriptor Records for the Backing Store

Record Type	*Fields*	*Format*	*Description*
bsp_when	t	P8	Specifies when ar.bsp is saved. The backing store pointer may be saved, along with the ar.bspstore pointer and the ar.rnat register, to indicate a discontinuity in the backing store.
bsp_gr	gr	P3	Specifies GR where ar.bsp is saved.
bsp_psprel	pspoff	P8	Specifies memory location where ar.bsp is saved, as a psp-relative offset.
bsp_sprel	spoff	P8	Specifies memory location where ar.bsp is saved, as an sp-relative offset.
bspstore_when	t	P8	Specifies when ar.bspstore is saved.
bspstore_gr	gr	P3	Specifies GR where ar.bspstore is saved.
bspstore_psprel	pspoff	P8	Specifies memory location where ar.bspstore is saved, as a psp-relative offset.
bspstore_sprel	spoff	P8	Specifies memory location where ar.bspstore is saved, as an sp-relative offset.
rnat_when	t	P8	Specifies when ar.rnat is saved.
rnat_gr	gr	P3	Specifies GR where ar.rnat is saved.
rnat_psprel	pspoff	P8	Specifies memory location where ar.rnat is saved, as a psp-relative offset.
rnat_sprel	spoff	P8	Specifies memory location where ar.rnat is saved, as an sp-relative offset.

Rules for using unwind descriptors

Preserved registers that are saved in the prologue region must be specified with one or more of the following descriptor records:

- prologue_gr (rp, ar.pfs, psp, and the predicates).
- mem_stack_v (psp is saved in a GR).
- rp_when, rp_gr, rp_psprel, or rp_sprel (rp).
- pfs_when, pfs_gr, pfs_psprel, or pfs_sprel (ar.pfs).
- unat_when, unat_gr, unat_psprel, or unat_sprel (ar.unat).
- lc_when, lc_gr, lc_psprel, or lc_sprel (ar.lc).
- fpsr_when, fpsr_gr, fpsr_psprel, or fpsr_sprel (ar.fpsr).
- fr_mem, frgr_mem, or gr_mem (FRs and GRs).
- br_mem or br_gr (BRs).

If a preserved register is not named by one or more of these records, it is assumed that the prologue does not save or modify that register.

The locations where preserved registers are saved are determined as follows:

1. Certain descriptor records explicitly name a save location for a register (records whose names end with "_gr," "_psprel," or "_sprel"). If a register is described by one of these records, the unwinder uses the named location.
2. Some descriptor records specify that registers are saved to the spill area (fr_mem, frgr_mem, gr_mem, br_mem). These locations are determined by the conventions for the spill area.
3. Any remaining registers that are named as saved, but do not have an explicit save location, are assigned consecutive GRs, beginning with the GR identified by the prologue_gr region header record. If the prologue region uses a prologue header record, the first GR is assumed to be GR 32. The registers are saved as needed in the following order:
   a. Return pointer, rp.
   b. Previous function state, ar.pfs.
   c. Previous stack pointer, psp.
   d. Predicates.
   e. User NaT collection register, ar.unat.
   f. Loop counter, ar.lc.
   g. Floating-point status register, ar.fpsr.
   h. Primary unat collection.

   Note that the only way to indicate that any of the last four groups of registers are saved, without explicitly specifying a save location, is to use one of the corresponding _when descriptor records.

Descriptor records for body regions

The following table lists the optional descriptor records that may be used to describe body regions. In the absence of these descriptors, a body region is assumed to inherit its entry state from the previous region.

Table 11–12. Body Region Descriptor Records

Record Type	*Fields*	*Format*	*Description*
epilogue	t, ecount	B2/B3	Body region contains epilogue code for one or more prologues.
label_state	label	B1/B4	Labels the entry state for future reference.
copy_state	label	B1/B4	Use labeled entry state as entry state for this region.

- **t** indicates the location of the instruction that restores the previous sp value, relative to the end of the region. The number is a count of the remaining instruction slots to the end of the region (thus, a value of 0 indicates the final slot in the region).
- **ecount** indicates how many additional levels of nested shrink-wrap regions are being popped at the end of a body region with epilogue code. A value of 0 indicates that one level should be popped.

- **label** identifies a previously-specified body region, whose entry state should be copied for this body region.

Descriptor records for body or prologue regions

This section lists the descriptor records that may be used to describe either prologue or body regions. These descriptors provide complete generality for compilers to perform register spills and restores anywhere in the procedure, without creating an arbitrary boundary between prologue and body.

Table 11–13. General Unwind Descriptors

Record Type	*Fields*	*Format*	*Description*
spill_psprel	t, reg, pspoff	X1	Specifies when and where reg is saved, as a psp-relative offset.
spill_sprel	t, reg, spoff	X1	Specifies when and where reg is saved, as an sp-relative offset.
spill_reg	t, reg, treg	X2	Specifies when and where reg is saved in another register, treg, or restored.
spill_psprel_p	qp, t, reg, pspoff	X3	Specifies when and where reg is saved, as a psp-relative offset, under predicate qp.
spill_sprel_p	qp, t, reg, spoff	X3	Specifies when and where reg is saved, as an sp-relative offset, under predicate qp.
spill_reg_p	qp, t, reg, treg	X4	Specifies when and where reg is saved in another register, treg, or restored, under predicate qp.

- **reg** identifies the register being spilled or restored at the given point in the code. This field may indicate any of the preserved GRs, FRs, BRs, ARs, predicates, previous sp, primary unat, or return pointer. See Appendix B for the encoding of this field.
- **treg** identifies a target register to which the value being spilled is copied. This field may indicate any GR, FR, or BR; it may also contain the special "Restore" target, indicating the point at which a register is restored. See Appendix B for the encoding of this field.
- **qp** identifies a qualifying predicate, which determines whether the indicated spill or restore instruction executes. The qualifying predicate must be a preserved predicate if there are any procedure calls in the range between the spill and restore, and it must remain live throughout the range.

Processing unwind descriptors

The unwind process for a frame begins by locating the unwind table entry for a given IP. If there is no unwind table entry, the unwinder should use the default conditions for this frame: leaf procedure, no memory stack frame, and no saved registers.

If there is an unwind table entry, the unwinder then locates the unwind information block and checks the size of the unwind descriptor area. If this area is zero length, the unwinder should use the default conditions as above.

In preparation for reading the unwind descriptor records, the unwinder should start with an initial current state record, and an empty stack of state records. A state record describes the locations of all preserved registers at entry to a region. The initial value of the current state record should describe the frame in its default conditions.

The unwind descriptor records should be read and processed sequentially, beginning with the first descriptor record for a procedure, continuing until the IP is contained within the current region. For each prologue region header, the current state record should be pushed on the stack, and the descriptor records for the prologue region should be applied to the current state record. When a body region with epilogue code is seen, one or more states should be popped from the stack, and the entry state for the next region is taken as the last state popped. This restores the current state to the entry state of the matching prologue.

When the current IP is within a body region, the unwinder can generate the context of the previous frame by restoring registers as indicated by the current state record. If the body region has epilogue code, and the IP is beyond the indicated point where sp is restored, the unwinder should assume that sp has already been restored, and that all registers spilled to the memory stack frame except those between psp and psp+16 have also been restored. Registers spilled to the scratch area in the caller's frame may not have been restored at that point, and the unwinder should use the values in memory.

When the current IP is within a prologue region, the unwinder must look for descriptor records that specify a time parameter that is at or beyond the current IP. It should ignore these state modifications when applying descriptor records to the current state. If a register is saved but does not have a specified time, the unwind may assume that the original value is not destroyed within the prologue, so it may ignore it.

The layout and size of the preserved register spill area cannot be determined without reading all the prologue region descriptor records in the procedure, and merging the save masks for the general registers, floating-point registers, and branch registers.

11.4.3 Language-specific data area

The try region table for C++ could be divided into two parts: a try|catch *table and a cleanup action table. As illustrated in Figure 11–2, the table consists of two 32-bit integers followed by the two tables. The first field,* tcnt, *contains the number of* try|catch *table entries, and the second field,* ccnt, *contains the number of cleanup action table entries. The* try|catch *table consists of a list of four-word entries, sorted by the region end address. The first two words of each entry identify the starting and ending addresses of the region, the third word points to the catch clause, and the fourth word points to the exception handler. The cleanup action table consists of a list of three-word entries, also sorted by the region end address. The first two words of each entry identify the starting and ending addresses of the region, and the third word points to a list of cleanup actions.*

Chapter 12

Dynamic Linking

12.1 Position-independent code

All code conforming to these conventions must be position independent (PIC). This allows their text segments to remain pure so they can be shared among many processes. Position-independence imposes two requirements on generated code:

- Code that forms an absolute address referring to any address in the load module's text or data segments is not allowed, since the code would have to be relocated at load time, making it non-sharable. All branches must be pc-relative, references to constants and literals in the text segment must be either pc-relative or indirect via the linkage table, and references to the data segment must be relative to a base register (typically `gp`).
- Code that references symbols that are or may be imported from other load modules must use indirect addressing through a linkage table. The linker is expected to resolve procedure calls by creating import stubs, but the compilers must generate indirect loads and stores for data items that may be dynamically bound. In both cases, the indirection is made through the linkage table, allocated by the linker, and initialized by the dynamic loader; the linkage table is described below.

Procedure calls and long branch stubs

Normal procedure calls can be made with the `br.call` instruction, which uses pc-relative addressing. There are three possible cases at link time:

- If the target is not within the same load module, or if it is subject to pre-emption by an earlier definition from another load module, the linker must allocate an import stub and resolve the `br.call` instruction to the stub.
- If the target is known to be within the same load module and the displacement is small enough, this instruction can be statically resolved to the call target.
- If the target is within the same load module, but the displacement is too large for the `br.call` instruction, the linker must allocate a long branch stub, as described in Section 8.4. The long branch stub itself must satisfy the PIC requirements. If the target is within range of the stub, the stub may use a pc-relative `br` instruction; otherwise, it must load the address of the target from the linkage table.

Access to the data segment

The DLL's short data segment must be accessed through the `gp` register, which is defined to point to the short data segment on entry to any DLL procedure. The `gp` register is used to access both the linkage tables and statically-allocated data. The DLL's long data segments must be accessed via the linkage table.

There are several cases here:

- Global variables that are imported from another load module, or that are subject to pre-emption by an earlier definition in another load module, must be accessed indirectly through the linkage table. The compiler must generate code to load a pointer from the linkage table, using gp-relative addressing, then access the data item using that pointer. The compiler does not have to allocate the linkage table; there are relocations defined in the object file format that instruct the linker to allocate a linkage table slot and supply the gp-relative address of that slot.
- Small, statically-allocated variables of local scope, or global variables whose definitions are not subject to pre-emption, may be placed in the short data segment and accessed directly with gp-relative addressing.
- Large variables, regardless of scope or pre-emption, must be placed in a long data segment, and accessed via the linkage table or pointer table.

The partitioning of the data into the short and long data segments is described in Section 3.2.

Access to constants and literals in the text segment

Constants and literals allocated in the text segment should be accessed with pc-relative addressing, or with indirect addressing via the linkage table.

Materializing function pointers

Function pointers must be materialized by loading a word from the data segment. They may not be materialized from immediate operands.

12.2 Import stubs

When the linker determines that a procedure call refers to an entry point in a different load module, it resolves the reference locally by building an import stub with the same name as the intended target. The import stub contains code that obtains the entry point and gp value from the linkage table, then transfers control, as described in Section 8.4.

If the compiler is provided with enough information to know that a particular entry point is in a different load module, it may generate a calling sequence that obviates the need for the linker to build an import stub. This calling sequence, however, is ABI specific, and is not specified in this document.

12.3 The dynamic loader

The dynamic loader is a component of the operating system software that locates all the load modules belonging to an application, loads them into memory, and binds the symbolic references among them. Most of the operation of the dynamic loader is specific to the particular operating system environment, and is further described in the ABIs for those environments. The common runtime architecture has been designed to minimize the amount of work involved in the binding process, by concentrating most of the relocation required in the linkage tables, and by prohibiting any items in the text segment that may require dynamic relocation.

Chapter 13

System Interfaces

13.1 Program startup

An application begins its execution at a specified program entry point, which depends on the primary language in which the application is written. For C programs, the function `main` is the program entry point. On most operating systems, however, some system-dependent initialization must take place before control is transferred to this entry point. This initialization may take place in the OS or in the DLL loader.

This section presents a general overview of what an application expects when its program entry point receives control. The ABI document for each operating system is expected to contain the details.

13.1.1 Initial memory stack

The memory stack pointer, `sp`, must be properly aligned, and must contain an address that is suitable for allocation of the program's first stack frame. There must be a 16-byte scratch area available for use, beginning at the address in `sp`, but the application may make no further assumptions about the contents of the memory stack beyond the scratch area.

13.1.2 Initial register values

The `sp` and `gp` registers must be initialized correctly, `sp` as described above and `gp` to the global pointer value for the main program's short data segment.

The floating-point status register should be initialized as shown in Table 13–1. The global trap disable bits (`ar.fpsr` bits 0–5) should all be initialized to ones.

Table 13–1. Initial Value of the Floating-Point Status Register

Status Field	*Flags*	*td*	*rc*	*pc*	*wre*	*ftz*
sf0	000000	0	00	11	0	0
sf1	000000	1	00	11	1	0
sf2 and sf3	000000	1	00	11	0	0

The initial stack frame must be setup with 0 input and local registers, and at least 4 output registers (as if the program entry point had been called with at

least four parameters). The contents of the parameter registers, in0–in7, are system-dependent, and are typically used for transmitting the program arguments.

13.2 System calls

System API routines are called using the standard calling conventions described in Chapter 8. Any special interfaces between these API routines and the operating system itself is system-dependent, and these API routines are typically supplied in a system DLL.

13.3 Traps and signals

When the OS delivers a signal or an exception to a user process, it must make the following available to the process:

- A context record, containing the full user-visible context, as described in Chapter 10.
- The cause of the trap. If the trap was caused by an instruction, the information must be sufficient to identify the bundle and slot.

When a trap or signal handler returns, OS help is necessary for restoring the complete context (via RFI). Thus, the OS must build a dummy stack frame for the handler, so that a return from the handler will transfer to an OS entry point that can restore the full context (this is sigreturn on HP-UX).

The OS must provide a new 16-byte scratch area prior to the stack frame created for the signal handler, so that the scratch area belonging to the interrupted procedure is not disturbed during signal processing.

The OS must also set the floating-point status register to the initial value specified in Table 13–1 prior to delivering a signal or exception.

Trap handlers will often need to look at the state of the registers at the time of the trap. Since the dynamic general registers are all hidden in the register stack backing store in memory, the application may need to perform some careful calculations to obtain access to the values of these registers. In addition, the OS may deliver a context in which the backing store is split into two non-contiguous areas. The OS-specific runtime library should provide an API routine to build an image of the dynamic registers from the context record.

Appendix A

Standard Header Files

A.1 Implementation Limits

The following constants are defined in the <limits.h> header file.

```
#define CHAR_BIT                8
#define SCHAR_MIN               (–128)
#define SCHAR_MAX               127
#define UCHAR_MAX               255

/* MB_LEN_MAX determined by locale information */

#define CHAR_MIN                SCHAR_MIN
#define CHAR_MAX                SCHAR_MAX

#define SHRT_MIN                (–32768)
#define SHRT_MAX                32767
#define USHRT_MAX               65535

#define INT_MIN                 (–2147483647–1)
#define INT_MAX                 2147483647
#define UINT_MAX                4294967295

#define __INT64_MIN             (–9223372036854775807–1)
#define __INT64_MAX             9223372036854775807
#define __UINT64_MAX            18446744073709551615
```

A.2 Floating-Point Definitions

The following constants are defined in the <float.h> header file.

```
#define FLT_DIG                 6       /* Max (decimal) digits of prec. */
#define FLT_EPSILON             1.19209290E–07F
#define FLT_MANT_DIG            24
#define FLT_MAX                 3.40282347E+38F
#define FLT_MAX_10_EXP          38
#define FLT_MAX_EXP             128
#define FLT_MIN                 1.17549435E–38F
#define FLT_MIN_10_EXP          (–37)
#define FLT_MIN_EXP             (–125)
#define FLT_RADIX               2
```

```
#define FLT_ROUNDS              1
#define FLT_GUARD               0
#define FLT_NORMALIZE           0

#define DBL_DIG                 15      /* Max (decimal) digits of prec. */
#define DBL_EPSILON             2.2204460492503131E–16
#define DBL_MANT_DIG            53
#define DBL_MAX                 1.7976931348623157E+308
#define DBL_MAX_10_EXP          308
#define DBL_MAX_EXP             1024
#define DBL_MIN                 2.2250738585072014E–308
#define DBL_MIN_10_EXP          (–307)
#define DBL_MIN_EXP             (–1021)

#define EXT_DIG                 18      /* Max (decimal) digits of prec. */
#define EXT_EPSILON             1.08420217248550443400 75E–19W
#define EXT_MANT_DIG            64
#define EXT_MAX                 1.18973149535723176502e+4932W
#define EXT_MAX_10_EXP          (+4932)
#define EXT_MAX_EXP             (+16384)
#define EXT_MIN                 3.36210314311209350626e–4932W
#define EXT_MIN_10_EXP          (–4931)
#define EXT_MIN_EXP             (–16381)

#define QUAD_DIG                33      /* Max (decimal) digits of prec. */
#define QUAD_EPSILON 1.92592994438723585305597794258492732E–34Q
#define QUAD_MANT_DIG           113
#define QUAD_MAX 1.189731495357231765085759326628007 02E+4932Q
#define QUAD_MAX_10_EXP         (+4932)
#define QUAD_MAX_EXP            (+16384)
#define QUAD_MIN 3.362103143112093506262677817321752 60E–4932Q
#define QUAD_MIN_10_EXP         (–4931)
#define QUAD_MIN_EXP            (–16381)
```

A.3 Variable Argument List Macros

The following definitions roughly define the operation of the variable argument list macros provided in the <stdarg.h> header file. Similar definitions for K&R C may be found in <varargs.h>.

```
typedef char *va_list;

#define _VA_ALIGN(list, align)                                          \
        (va_list)(((unsigned int)(list) + (align) – 1) & ~((align) – 1))

#define va_start(list, parmN)   (list = (va_list)(&parmN + 1))

#ifdef __LITTLE_ENDIAN__

#define va_arg(list, mode) (                                            \
          list = _VA_ALIGN(list, ((sizeof(mode) > 8) ? 16 : 8)),        \
          *(mode *)list++                                               \
          )

#else /* __BIG_ENDIAN__ */
```

```
#define va_arg(list, mode) (                                                \
        list = _VA_ALIGN(list, ((alignof(mode) > 8) ? 16 : 8)) +            \
                ( ((sizeof(mode) < 8) && !__is_aggregate(mode)) ?           \
                        8 - sizeof(mode) : 0 ),                             \
        *(mode *)list++                                                     \
        )

#endif /* __BIG_ENDIAN__ */
```

The big endian version of the va_arg macro requires built-in alignof() and isaggregate() functions in the compiler; the latter returns true if the type given as the argument is an aggregate type.

A.4 Setjmp/Longjmp

The following definition is provided in the <setjmp.h> header file.

```
typedef __float80 jmp_buf[_JBLEN];
```

The jump buffer must be long enough to contain the context defined in Section 10.3, and should include additional space reserved for future use. It must be declared to guarantee 16-byte alignment (for example, as an array of __float80). Its contents include the following registers:

- Instruction address (ip)—the return BR from the call to setjmp()
- Stack pointer (sp)
- Frame state—the ar.pfs register from the call to setjmp()
- Backing store pointer (ar.bsp)
- General registers r4–r7
- NaT bits for general registers r4–r7 (shifted to a consistent position independent of the jump buffer address)
- Floating-point registers f2–f5 and f16–f31
- Floating-point status register (ar.fpsr)
- Predicates p1–p5 and p16–p63
- Branch registers b1–b5
- User NaT collection register (ar.unat)
- RSE NaT collection register (ar.rnat)
- Loop counter (ar.lc)
- Signal mask (for sigsetjmp/siglongjmp)

Note that the epilog counter (ar.ec) is automatically preserved with the ar.pfs register.

The jump buffer contents should also include a “signature” to identify its version number and architecture for compatibility with future hardware and software releases.

The size of the jump buffer (the value of _JBLEN) and the locations of individual items within the jump buffer are ABI specific.

Appendix B

Unwind Descriptor Record Formats

B.1 Overview

The unwind descriptor records are encoded in variable-length byte strings. The various record formats are described in this appendix.

The first byte of each record is sufficient to determine its format. The high-order bit of this byte determines whether it is a header record (if the bit is zero), or a region descriptor record (if the bit is one). The remaining bits and any subsequent bytes are divided into separate fields. In most formats, the first field, r, identifies the record type. The record formats are listed by the bit pattern of the first byte in Table B–1.

Table B–1. Record Formats

Region Header Records		*Prologue Descriptor Records*		*Body Descriptor Records*	
Bit Pattern	***Format***	***Bit Pattern***	***Format***	***Bit Pattern***	***Format***
00-- ----	R1	100- ----	P1	10-- ----	B1
0100 0---	R2	1010 ----	P2		
0110 00--	R3	1011 0---	P3		
		1011 1000	P4		
		1011 1001	P5		
		110- ----	P6	110- ----	B2
		1110 ----	P7	1110 0000	B3
		1111 0000	P8	1111 -000	B4
		1111 0001	P9		
		1111 1001	X1	1111 1001	X1
		1111 1010	X2	1111 1010	X2
		1111 1011	X3	1111 1011	X3
		1111 1100	X4	1111 1100	X4
		1111 1111	P10		

Some fields in the unwind descriptor records are variable in length. The variable-length encoding uses the ULEB128 (Unsigned Little-Endian Base 128) encoding, described below:

- Divide the number into groups of 7 bits, beginning at the low-order end.
- Discard all groups of leading zeroes, but keep at least the first (low-order) group if the number is all zeroes.
- Place a 1 bit to the left of of all but the last group; place a 0 bit to the left of the last group. This forms one or more 8-bit groups.

The following table shows example ULEB128 encodings for several numbers:

Table B–2. Example ULEB128 Encodings

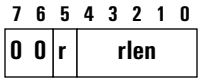
Value	*Encoding*	*Interpretation*
0	`00000000`	0
127	`01111111`	127
128	`10000000 00000001`	0 + (1 << 7)
1544	`10001000 00001100`	8 + (12 << 7)
49,802	`10001010 10000101 00000011`	10 + (5 << 7) + (3 << 14)

Fields in the ULEB128 format always follow the fixed fields, and begin on a byte boundary.

B.2 Region Header Records

The prologue and body region header records can appear in either format R1 or R3, depending on the magnitude of the region length field. If the region length is no greater than 31 instructions, the R1 format may be used; otherwise, format R3 must be used.

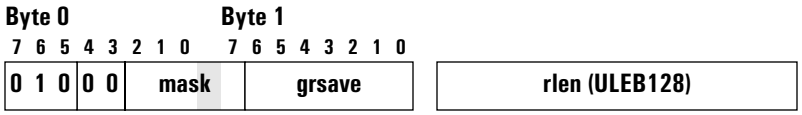
Format R1



This format is used for the short forms of the prologue and body region header records. The r bit identifies the record type, as shown in the following table:

Record Type	*r*
prologue	0
body	1

Format R2

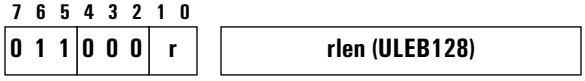


This format is used only for the prologue_gr region header record. The following table shows the meaning of the bits in the mask field:

Mask bit	*Meaning when bit is set*
byte 0, bit 2	rp is saved in standard GR
byte 0, bit 1	ar.pfs is saved in standard GR
byte 0, bit 0	psp is saved in standard GR
byte 1, bit 7	predicates are saved in standard GR

The grsave field identifies the general register in which the first of these values is stored. Additional general registers are used as needed. For example, assume that rp, ar.pfs, and the predicates are stored, but not psp. The mask bits would be 1101, and grsave might be set to 39, indicating that the three values are stored in r39, r40, and r41, respectively.

Format R3

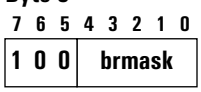


This format is used for the long forms of the prologue and body region header records. The r field identifies the record type, as shown in the following table:

Record Type	*r*
prologue	00
body	01

B.3 Descriptor Records for Prologue Regions

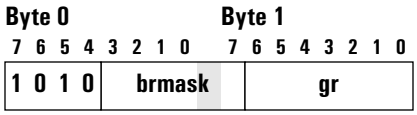
Format P1



This format is used only for the br_mem descriptor record.

The five bits in the brmask field are used to indicate which of the five preserved branch registers (b1–b5) are saved in the prologue. Bit 0 corresponds to b1; bit 4 corresponds to b5. If the bit is clear, the corresponding register is not saved; if the bit is set, the corresponding register is saved.

Format P2

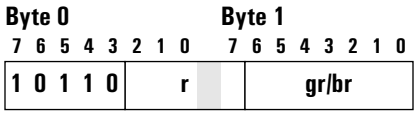


This format is used only for the br_gr descriptor record.

The five bits in the brmask field are used to indicate which of the five preserved branch registers (b1–b5) are saved in the prologue. Bit 7 of byte 1 corresponds to b1; bit 3 of byte 0 corresponds to b5. If the bit is clear, the corresponding register is not saved; if the bit is set, the corresponding register is saved.

The gr field identifies the general register in which the first of these registers is stored. Additional general registers are used as needed. For example, assume that b1, b4, and b5 are stored. The mask bits would be 11001, and gr might be set to 37, indicating that the three branch registers are stored in r37, r38, and r39, respectively.

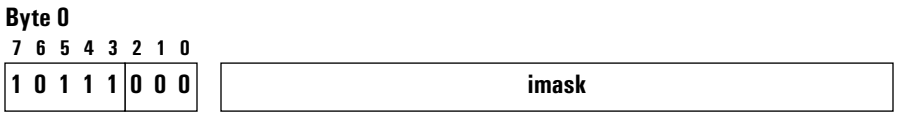
Format P3



This format is used by the group of descriptor records that specify a GR or BR number. The record type is identified by the r field, which is read as a four bit number whose low-order bit is bit 7 of byte 1. The following table shows the record types:

Record Type	*r*	*Record Type*	*r*
psp_gr	0	rp_br	6
rp_gr	1	rnat_gr	7
pfs_gr	2	bsp_gr	8
preds_gr	3	bspstore_gr	9
unat_gr	4	fpsr_gr	10
lc_gr	5	priunat_gr	11

Format P4



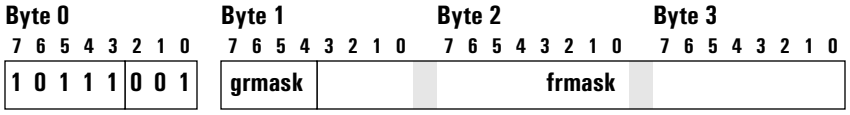
This format is used only by the spill_mask descriptor record. The first byte is followed by the imask field, whose length is determined by the length of the current prologue region as given by the region header record. The imask field contains two bits for each instruction slot in the region, and the size is rounded up to the next whole number of bytes, if necessary.

The high-order (leftmost) two bits of the first byte of the imask field correspond to the first instruction slot of the region. Bit pairs are read from left to right (high-order bits to low-order bits) within each byte, and bytes are read

from increasing memory addresses. Each bit field describes the behavior of the corresponding instruction slot as follows:

Bit Pair	*Meaning*
00	The instruction slot does not save one of these registers
01	the instruction slot saves the next floating-point register
10	the instruction slot saves the next general register
11	the instruction slot saves the next branch register

Format P5



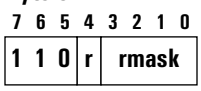
This format is used only by the frgr_mem descriptor record.

The bits in the grmask field correspond to the preserved general registers (r4–r7). The bits are read from right to left: bit 4 of byte 1 corresponds to r4, and bit 7 corresponds to r7.

The bits in the frmask field correspond to the preserved floating-point registers (f2–f5 and f16–f31). The bits are read from right to left: bit 0 of byte 3 corresponds to f2, and bit 3 of byte 1 corresponds to f31.

A value of 1 in each bit position indicates that the corresponding register is saved.

Format P6

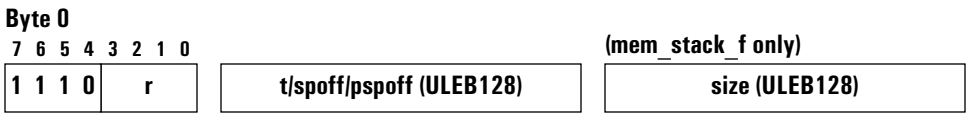


This format is used by the fr_mem and gr_mem descriptor records. The r bit identifies the record type, as shown in the following table:

Record Type	*r*
fr_mem	0
gr_mem	1

The bits in the rmask field correspond to either the preserved general registers (r4–r7) or the set of the first four preserved floating-point registers (f2–f5). The bits are read from right to left: bit 0 corresponds to r4 or f2, and bit 3 corresponds to r7 or f5. A value of 1 in each bit position indicates that the corresponding register is saved.

Format P7

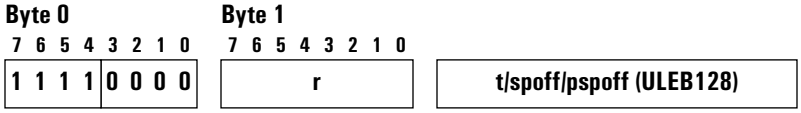


This format is used for a number of descriptor records. The r field identifies the record type, as shown in the following table:

Record Type	*r*	*Additional ULEB128 Fields*
mem_stack_f	0	t, size
mem_stack_v	1	t
spill_base	2	pspoff
psp_sprel	3	spoff
rp_when	4	t
rp_psprel	5	pspoff
pfs_when	6	t
pfs_psprel	7	pspoff
preds_when	8	t
preds_psprel	9	pspoff
lc_when	10	t
lc_psprel	11	pspoff
unat_when	12	t
unat_psprel	13	pspoff
fpsr_when	14	t
fpsr_psprel	15	pspoff

Stack pointer offsets (spoff) are represented as positive word offsets from the top of the stack frame (i.e., the location is sp + 4 * spoff). Previous stack pointer offsets (pspoff) are encoded as positive numbers representing a negative word offset relative to psp+16 (i.e., the location is psp + 16 – 4 * pspoff).

Format P8



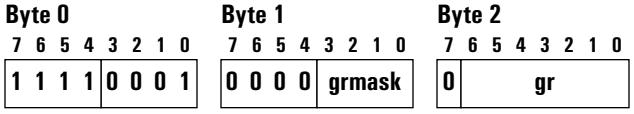
This format is used for a number of descriptor records. The r field identifies the record type, as shown in the following table:

Record Type	*r*	*Additional ULEB128 Fields*
rp_sprel	1	spoff
pfs_sprel	2	spoff
preds_sprel	3	spoff
lc_sprel	4	spoff

Record Type	*r*	*Additional ULEB128 Fields*
unat_sprel	5	spoff
fpsr_sprel	6	spoff
bsp_when	7	t
bsp_psprel	8	pspoff
bsp_sprel	9	spoff
bspstore_when	10	t
bspstore_psprel	11	pspoff
bspstore_sprel	12	spoff
rnat_when	13	t
rnat_psprel	14	pspoff
rnat_sprel	15	spoff
priunat_when_gr	16	t
priunat_psprel	17	pspoff
priunat_sprel	18	spoff
priunat_when_mem	19	t

Stack pointer offsets (spoff) are represented as positive word offsets from the top of the stack frame (i.e., the location is sp + 4 * spoff). Previous stack pointer offsets (pspoff) are encoded as positive numbers representing a negative word offset relative to psp+16 (i.e., the location is psp + 16 – 4 * pspoff).

Format P9

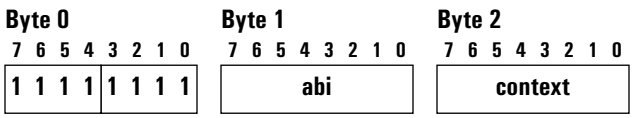


This format is used only by the gr_gr descriptor record.

The bits in the grmask field correspond to the preserved general registers (r4–r7). The bits are read from right to left: bit 0 of byte 1 corresponds to r4, and bit 3 corresponds to r7.

The gr field identifies the general register in which the first of these registers is stored. Additional general registers are used as needed. For example, assume that r4, r5, and r7 are stored. The mask bits would be 1011, and gr might be set to 37, indicating that the three preserved general registers are stored in r37, r38, and r39, respectively.

Format P10



This format is reserved for ABI-specific unwind descriptor records, typically to identify a region whose stack frame indicates some saved context record (e.g., a Unix signal context).

The values currently defined for the abi field are shown in the following table:

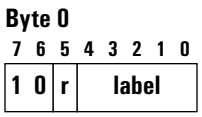
Value	*ABI*
0	Unix SVR4
1	HP-UX
2	Windows NT

The interpretation of the context field is ABI dependent.

B.4 Descriptor Records for Body Regions

The epilogue, label_state, and copy_state descriptor records can each appear in two formats, depending on the magnitudes of their fields.

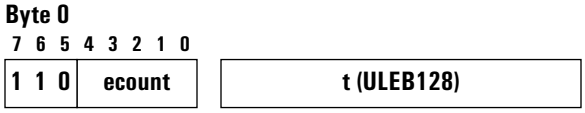
Format B1



This record is used for the short form of label_state and copy_state descriptor records. If the label is no greater than 31, this format may be used; otherwise, format B4 must be used. The record types are shown in the following table:

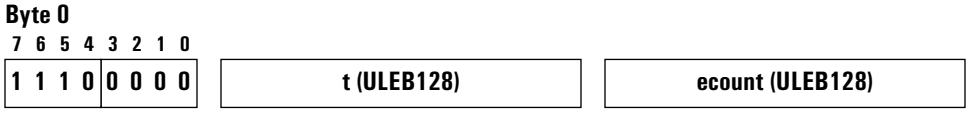
Record Type	*r*
label_state	0
copy_state	1

Format B2



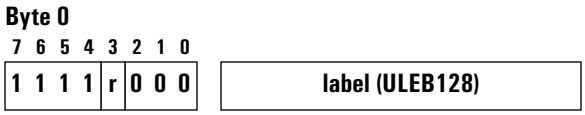
This format is used only for the short form of the epilogue descriptor record. If the ecount field is no greater than 31, this format may be used; otherwise, format B3 must be used.

Format B3



This format is used only for the long form of the epilogue descriptor record.

Format B4



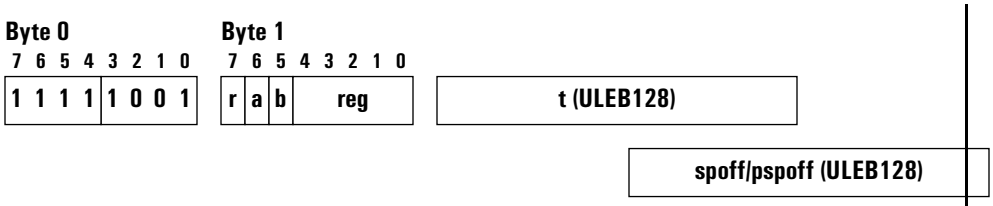
This format is used only for the long form of the label_state and copy_state descriptor records. The record types are shown in the following table:

Record Type	*r*
label_state	0
copy_state	1

B.5 Descriptor Records for Body or Prologue Regions

The record formats listed here describe general spills and restores, and may appear in either body or prologue regions.

Format X1



This format is used by the spill_psprel and spill_sprel descriptor records, which identify when a register is saved by spilling to the memory stack. The r bit identifies the record type, as shown in the following table:

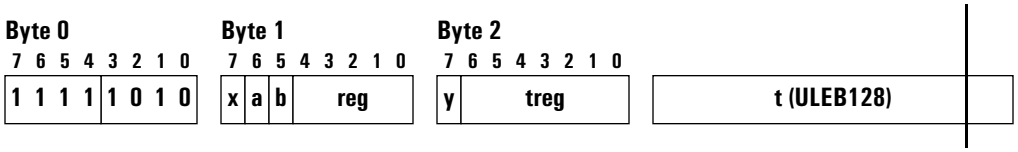
Record Type	*r*
spill_psprel	0
spill_sprel	1

The a, b, and reg fields identify the register being spilled. The encodings are given in the following table:

Record Type	*a*	*b*	*reg*
GR 4–7	0	0	gr
FR 2–5 or 16–31	0	1	fr
BR 1–5	1	0	br
Predicates	1	1	0
psp	1	1	1
priunat	1	1	2
rp	1	1	3
ar.bsp	1	1	4
ar.bspstore	1	1	5

Record Type	*a*	*b*	*reg*
ar.rnat	1	1	6
ar.unat	1	1	7
ar.fpsr	1	1	8
ar.pfs	1	1	9
ar.lc	1	1	10

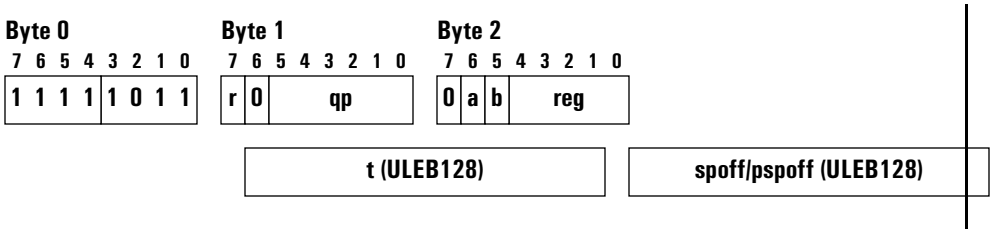
Format X2



This format is used only by the spill_reg descriptor record, which identifies when a register is saved by copying to another register, or when a register is restored from its spill location. The register being saved or restored is identified by the a, b, and reg fields, using the same encodings given above for Format X1. The target register to which the saved register is copied is identified by the x, y, and treg fields; a special encoding also indicates the "restore" operation. The encodings for these fields are given in the following table:

Record Type	*x*	*y*	*treg*
Restore	0	0	0
GR 1–127	0	0	gr
FR 2–127	0	1	fr
BR 0–7	1	0	br

Format X3

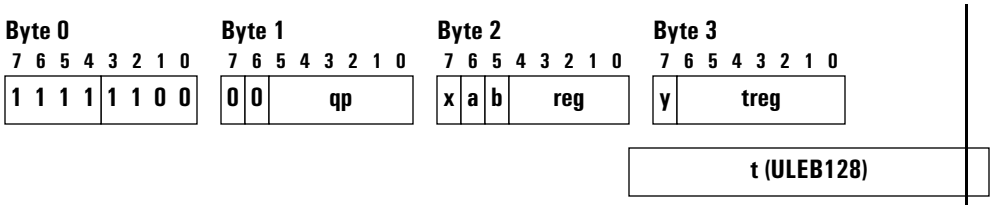


This format is used by the spill_psprel_p and spill_sprel_p descriptor records, which identify when a register is saved under control of a predicate. The r bit identifies the record type, as shown in the following table:

Record Type	*r*
spill_psprel_p	0
spill_sprel_p	1

The qp field identifies the controlling predicate. The remaining fields are encoded the same as Format X1.

Format X4



This format is used only by the spill_reg_p descriptor record, which identifies when a register is saved to another register under control of a predicate, or when a register is restored under control of a predicate.

The qp field identifies the controlling predicate. The remaining fields are encoded the same as Formats X1 and X2.